EPIC-2

Workshop on Explicitly Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques

Istanbul, Turkey

November 18th, 2002

Workshop Proceedings

EPIC-2

Held in Conjunction with MICRO-35
Istanbul, Turkey
December 2, 2001

http://systems.cs.colorado.edu/EPIC2/

GENERAL CHAIRS

David August, Princeton University
Dan Connors, University of Colorado

PROGRAM CHAIRS

Carole Dulong, Intel Corporation
Rick Hank, Hewlett-Packard Corporation

PROGRAM COMMITTEE

Santosh Abraham, Sun Microsystems
David August, Princeton University
Brad Calder, University of California-San Diego
Dan Connors, University of Colorado
David Gillies Microsoft Research
Tom Conte, North Carolina State University
Jim Dehnert, Transmeta Corporation
Kemal Ebcioglu, IBM Research
Wen-mei Hwu, University of Illinois
Suneel Jain, Hewlett-Packard Corporation
Teresa Johnson, Hewlett-Packard Corporation
Vinod Kathail, Hewlett-Packard Research Labs
Dan Lavery, Intel Corporation
Scott Mahlke, University of Michigan
Krishna Palem, Georgia Tech University
Jim Pierce, Intel Corporation
Nancy Warter, Cal State- Los Angeles

John W. Sias

IMPACT Research Group
University of Illinois at Urbana-Champaign
Urbana, IL 61801
sias@crhc.uiuc.edu

The IMPACT compiler has for several years provided a simulation-based platform for general EPIC compiler and architecture research, focusing especially on developing instruction-level parallelism using predication, control speculation, and aggressive code analysis. The advent of the Intel Itanium Architecture provided the first opportunity to demonstrate the efficacy of these techniques in a real, general-purpose hardware environment. The concurrent acceptance of more complex and control-intensive benchmarks (SPEC 2000) and a 64-bit addressing model presented additional issues for EPIC compilers. The process of applying the traditional strengths of the IMPACT compiler to this complex new environment has yielded a useful framework for the prototyping of new techniques on IPF, as well as many valuable insights about compilation and performance on Itanium and Itanium 2. We present an overview of the IMPACT-ia64 compilation environment, differentiating it from available production compilers. Working from data gathered from extensive microarchitectural performance monitoring features on SPEC Cint2000, we also present a discussion of a few key performance issues encountered in the compiler's development to date and their implications for future work.

Exploring and Optimizing Itanium2™ Cache(s) Performance for Scientific Computing

William Jalby
jalby@prism.uvsq.fr
PRiSM Laboratory
45 Avenue des Etats Unis
78035 Versailles Cedex France

Christophe Lemuet
lemuet@prism.uvsq.fr
PRiSM Laboratory
45 Avenue des Etats Unis
78035 Versailles Cedex France

Abstract

Memory hierarchies are a key component in getting high performance on modern microprocessors. To satisfy the ever increasing demand on data rate access, they are also becoming increasingly complex, Itanium2™'s cache system being a good example of this trend: three levels of caches, non blocking caches, high degree of parallelism (up to four memory access per cycle), sophisticated instructions for supporting prefetch and cache control etc …. Although all of these advanced features promise to offer large performance gains, in many cases, performance remains disappointing. In this paper, we study in detail performance behavior of simple scientific kernels (BLAS1) on an Itanium2™ cache system. We demonstrate, through systematic experimentations that performance can be very sensitive to the memory address stream structure, revealing that the underlying (hidden) cache organization (banking, load queues) has a major impact on performance. We develop specific instruction scheduling techniques allowing to reach excellent and very stable performance levels.

1. Introduction

Memory subsystem performance is essential to today's microprocessors [CS98]. Therefore, computer architects, have spent a large effort in inventing sophisticated mechanisms to improve data access rate (both in terms of latency and bandwidth) [FJ95, CB95]: multilevel caches, non blocking caches, out of order execution, prefetch instructions, etc …..

The good side of the story is that these mechanisms allow to offer excellent peak performance numbers. The bad side of the story is twofold: first, these mechanisms require from codes specific characteristics to reach peak performance [Ba95]: for example, caches need spatial/temporal locality, prefetch instructions require regular access stream. Second, the resulting complexity of
the memory subsystem is very high, one of the reason being that most of modern microprocessors simultaneously use a large number (if not all) of the mechanisms listed above. This last point does not only make design and fabrication difficult and expensive, but also, makes performance very sensitive to the interaction between these mechanisms themselves and the codes.

Previously, in the "old days" of vector machines, a similar trend was observed. To reach decent memory bandwidth, Cray XMP had a very complex memory subsystem organized in banks, sections, subsections, etc… The analysis of performance of even very simple codes was already very difficult [OL85].

In this paper, the Itanium2™ architecture was selected as a target for exploring cache performance. The Itanium2™ has an interesting cache system (multilevel, high degree of parallelism, sophisticated prefetch capabilities….) [IN02] combined with an original processor architecture (using aggressively static information provided by the compiler and speculative execution) [Ha00, SA00].

Due to the already complex nature of the problem, our study is currently restricted to the cache subsystem (excluding memory) and to simple vector codes (BLAS 1 type: Copy, Daxpy) yet fairly representative of memory address streams in scientific computing. The choice of scientific computing as a target application area is motivated by the excellent match between scientific codes (easy to analyze statically) and the Itanium2™ architecture (very well designed for exploiting static information).

Even with this rather limited scope in terms of application codes, our study reveals that performance behavior is rather complex and hard to analyze. Overlooked parameters such as relative positioning of

starting array address vis-à-vis cache line boundaries or page boundaries are shown to have major performance impact. In particular, the banking/interleaving structure of the L2 cache and the load/store queue structure are shown to have a major interaction with address stream, potentially inducing large performance loss.

We propose several techniques for scheduling Loads and Store Instructions, taking into account the bank structure of the L2 and the load/store queue mechanisms. These techniques are applied to the Copy and Daxpy kernels and experimental results validating the interest of such techniques are presented: optimal (or close to) performance levels are obtained and the performance itself does no longer suffer from instabilities due to the relative position of starting array address.

Section 2 describes our experimental setup: hardware platform as well as software platform (compiler and OS). In section 3, our target codes are presented. In Section 4, the experimental methodology is detailed: this covers parameter space description, measurement methodologies and results presentation/visualization. In Section 5, experimental results on simplified kernels are presented allowing a clear isolation of performance problems. In Section 6 (resp. 7), experimental results on Copy (resp. Daxpy) are presented and analyzed. Finally, in Section 8, a few concluding remarks and directions for future work are given.

2. Experimental setup

2.1. Hardware setup

The machine used in our experiments is a uniprocessor Itanium2™ based system equipped with 900Mhz processor and 3GB memory. The "general" processor architecture (an interesting combination of VLIW and advanced speculative mechanisms) is very well described in the literature [Ha00,SA00,Ba00,IN02]. Of particular importance in our study, is the fact that Itanium2™ is mostly an "in order processor": instructions are executed in the same order they are issued making instruction scheduling rather important. The only "major" exception to this rule is memory operation processing for which a partial out of order processing is allowed through the use of Load/Store queues.

The Itanium2™ offers a wide degree of parallelism:

- Six general purpose ALU, two integer units and one shift unit;
- The Data cache unit contains for memory ports allowing to service either four load requests or two load and two store requests in the same cycle;
- Two floating point Multiply Add units allowing to execute up two floating point multiply add operations per cycle;
- Six multimedia functional units;
- Three Branch units, ….

All of the computational units are fully pipelined, so each functional unit can accept one new instruction per clock cycle (in the absence of stalls).
Instructions are grouped together in blocks of three instructions (called a bundle). Up to two bundles can be issued per cycle. Due to the wealth of functional units, a rate of six instructions executed per cycle can be sustained. This has to be compared with the Itanium1™ processor where an important performance limitation was due to a shortage of memory access units. This shortage was very sensitive on memory bound computations.

On our test machine, caches are organized in three levels, all of them being on chip:

- L1 level split in a D cache (16KB) and an I cache (16KB), using 64 Bytes cache lines. However these caches cannot be used for storing floating point data;
- L2 level, unified, 256 KB, 8 way associative, uses a Write Back Allocate policy and a 128 Bytes cache line;
- L3 level, unified, 1.5 MB, 12 way associative, uses a 128 Bytes cache lines.

Latencies and bandwidth (for the floating point load/store instructions) of the various levels are given in the table below:

	Bandwidth (FP)	Latency (FP)
L1D	NA	NA
L2	32Bytes/cycle READ, or 16Bytes/cycle READ and 16Bytes/cycle WRITE	6 cycles minimum
L3	32B / cycles	12 cycles minimum

The L2 is capable of supporting up to four memory accesses per cycle. The L2 cache is organized in 16 banks, with an interleaving of 16Bytes: address 0 and 8 are located in bank 0, address 16 and 24 in bank 1, etc … The L2 cache non blocking nature is supported via a Load/Store queue (L2OzQ) capable of holding up to 32 operations which cannot satisfied by the L1D. This queue also allows making additional requests to the L2 while younger requests are blocked due to bank conflicts for example. In addition to bank conflicts other problems such as disambiguation of memory address have to be dealt with. A sequence of a Load and a Store addressing the same memory location should be detected and proper ordering should be enforced to preserve program

correction. Ideally, such a specific treatment should be triggered only by comparing the full address of Load and Store. Unfortunately if Loads and Stores execution are too close to each other, only a partial comparison (on the lower address bits) is carried out, generating potential performance penalties between memory references which have been improperly disambiguated. Finally, a maximum of 16 outstanding cache miss (request to L3 or memory) to unique cache lines can be handled by the L2 cache controller.

In addition to standard Load and Store instructions on floating point operands (single and double precision), the Itanium™ instruction set offers Load Floating Pair instruction capable of loading 16 Bytes at once, provided that the corresponding address are lined up on a 16 Bytes boundary.

2.2. Software environment

The test machine was running Linux IA-64 Red Hat 7.1 based on the 2.4.18 smp kernel. The page size used by the system was 16Kbytes.
Two different compilers were used:

- Intel® C++ Compiler Version 6.0, Build 20020614;
- Intel® C++ Compiler Version 7.0 Beta, Build 20020703.

The V6 version was used with –tpp2 flag to generate specific code for the Itanium2™.

On both compilers various options have been tested, however for our simple BLAS1 kernels, it was found that the combination of –O3 and –restrict was very powerful, fully using most of the advanced features of Itanium2™ architecture: software pipelining, prefetch instructions, predication, rotating registers… In getting top performance, the "–restrict" option was essential because it allowed the compiler to assume that distinct arrays were pointing to disjoint memory regions, therefore, allowing a full reordering of loads and stores. Fortran compiler was also tested, but again for our simple loops, the code generated was almost identical to the one obtained with C language.

Besides the compiler and OS, the perfmon toolkit, allowing direct access to various performance counters has been used.

3. Target codes

Two different types of kernels were used. The first one (called *memory stress kernels*) corresponds to artificial codes (i.e. they do not perform "useful" computations), the main goal being to explore cache system behavior. The second type corresponds to two typical *BLAS1 kernels* performing real useful computations.
In this paper results for two *memory stress kernels* are presented (identifiers X and Y refer to double precision floating point arrays):

- **Load/Load Load X(I), Load Y(I):** 2 "independent" load streams, no floating point arithmetic operations
- **Load/Store Load X(I), Store Y(I):** 1 load stream, 1 store stream, no floating point arithmetic operations

These two kernels were directly generated by hand in assembly code. Since they only consist of memory access instructions and simple address computation, we used systematically the mmi (or mmb) bundles. The goal of these kernels is first to test the maximum sustainable bandwidth and second to detect performance problems and third to develop workarounds. These kernels have been tested using different instruction ordering, either alternating between X and Y references (Load X(0), Load Y(0), Load X(1), Load Y(1), etc ….) or grouping (vectorizing) X and Y references (Load X(0), Load X(1), Load X(2), Load X(3), Load Y(0), Load Y(1), Load Y(2), Load Y(3) etc ….)

The two *BLAS1 kernels* used in this paper are simple vector loops:

- **Copy:** ***Y(I) = X(I)***, 1 load, 1 store streams ;
- **Daxpy:** ***Y(I) = Y(I) + a * X(I)***, 2 loads, 1 store streams but 1 load and the store stream referring to the same array Y;

For each of these two kernels, several source code variants hand generated (unrolled, vectorized) were developed, compiled and tested. However, only results for the simplest form (called Std) are reported since the compiler was already performing advanced optimizations and the performance gains of the other variants were negligible.

Following our study of the memory stress kernels, specific optimized versions of both Copy and Daxpy were generated by hand, incorporating the specific instruction scheduling techniques, which were found efficient on the memory stress kernels.

4. Measurement methodology

4.1. Parameter space

Besides the different kernels (described in the previous section) and their variants, other "major" parameters have been explored:

- **Operand location:** since we were interested in exploring cache performance (L2 and L3), two types of measurements were performed: the first one (called L2 region) in which all of the operands are located in L2, the second one (called L3 region) in which all of the operands are located in L3. This "localization" of the operands is achieved by a specific organization of the "*drive*r" code described later in Section 4.2.
- **Prefetch distance and mode:** for each of the loop variant, the compiler selected specific prefetch instruction type and distance. To analyze the impact of prefetch instruction type and prefetch distance, we modified directly in the assembly code the values chosen by the compiler.
- **Starting address of arrays:** in our experiments, the array layout in the virtual memory space is tightly controlled. In particular, the impact of the starting address of each array (X, Y) is studied in depth. To achieve this goal, the parameters Offset X (resp. Offset Y) are introduced, according to the following relations:

Element	Virtual address (Bytes)
X[i]	512K + Offset X + 8*i
Y[i]	512K + DIST1 + Offset Y + 8*i

DIST1 is an additional parameter mainly used to avoid array overlap, i.e. making sure that array X and Y are located in disjoint portions of memory space. In most of our experiments DIST1 remains constant, equal to 32 KB.

It could be argued that the starting address of arrays does not have a major impact on performance, however, as we will see, this is far from being true.

Systematic exploration of this parameter space is fairly expensive: with 2 arrays X and Y, offsets X and Y should both vary between 0 and 16K (the page size) with increment of 8 Bytes (corresponding to 2K values for offset X and 2K values for offset Y). All possible combinations between offset X and offset Y should be tested, leading to 2K x 2K = four millions of experiments!! To limit the combinatorial explosion in terms of experiments, the parameter space corresponding to offsets has been explored in two manners: first experiments were run with offsets varying between 0 and 512 with increments of 8 (spatial 2D exploration), second experiments were conducted with only one offset varying between 0 and 16KB, the other two being set to 0 (1D exploration). It should be noted that such a method could prevent us from discovering all of the pathological behavior. However all of the problems depending upon the relative value Offset X – Offset Y should be captured.

The term "*iteration count*" could become ambiguous, since in some variants unrolling is used. To avoid this problem, the term "*computation size*" is used where "*computation size*" denotes the total number of distinct elements of the array X accessed during the whole loop execution. Impact of the "*computation size*" was not directly studied: for example all of the problems arising with very short "*computation size*" were not tackled.

In our study, we mainly focused on steady state behavior, using a typical "*computation size*" of at least 1440 (corresponding to the computation of 1440 elements) which is large enough for reaching peak performance while still allowing us to keep the operands in the L2 cache.

Across the different variants and kernels, this "*computation size*" remains constant to allow a fair comparison.

4.2. Measurements

Measurements were performed on a stand-alone system (i.e. our benchmark code was the only user application running), only one measurement being performed at a time.

All of the timing measurements were performed using the *mov ar.itc* instruction to read the cycle counter of the processor itself.

Three different types of measurements were used:

1. Mes 1: a standard repetition loop surrounds the code/kernel to be tested. By using an appropriate array size, operands can be "kept" either in L2 or in L3; such a technique generate the well known curves with different plateaux corresponding to the different cache levels.
2. Mes 2: only one execution of the kernel is performed. For the L2 region measurements, the arrays are first loaded in the L2, and then the kernel is executed once. For the L3 measurements, the arrays are flushed from L2 (without being flushed from L3) before kernel execution.
3. Mes 3: several executions of the kernel are performed. In L2 region, the method is equivalent to Mes 1 while in L3 region, the Mes 3 method performs an "auto" flush of the arrays, by moving

through the array through the consecutive executions of the kernel.

While Mes1 and Mes3 modes can accommodate relatively low accuracy timers due to the repetition, Mes2 mode requires a fairly accurate timer.

For all of our experiments, the three types of measurement were performed and consistency between the three results obtained was systematically checked.

In addition, perfmon toolkit reading the various cache miss counters were used first to verify our assumptions that operands were effectively kept in the desired cache level and second that the penalties associated with DTLB remained negligible.

4.3. Results presentation

All of the performance numbers presented are normalized with respect to one iteration: i.e. the measurements correspond:

- To the average number of cycles to perform one pair of memory access: Load X(I) and Load Y(I) (resp Load X(I) and Store Y(I)) for the memory stress kernels;
- to the "average" number of cycles to compute one element of the result in the case of the BLAS1 kernels.

One of the major points of focus will be the impact of offsets on performance. Therefore, 2D plots (isosurface) will be systematically displayed. A "geographic" color code is used: dark blue corresponds to the best performance (lowest number of cycles) while dark red corresponds to the worst performance. These 2D plots will be very useful in understanding qualitatively the spatial nature of the phenomenon. Also, standard curves corresponding to cuts through the 2D plots will be presented to give a precise quantitative measure. The values for these cuts correspond to fixed values of offset Y. These values (offset Y = 0 and 392) were somewhat picked up arbitrarily, the main goal of these "cuts" is to provide precise performance numbers.

5. Memory stress kernels

For all the L2 region (resp. L3 region) 2D plots (figures 1 and 2) the same scale (and color code) is used. Similarly, the same vertical axis scale is used for the L2 region (resp. L3 region) curves.

5.1. Load Load results (Figure 1)

All of the kernels were unrolled 8 times first to minimize the impact of branch penalty and second to minimize the relative cost of prefetch instructions. Two types of Load/Load kernels were used, the first one called interleaved, the second one called vector optimized.

5.1.1. Load/Load Interleaved results

The corresponding interleaved version in assembly code is given below:

```
.b1_2:
 { .mmi
    ldfd    f32=[r32],64  // load x[8*i]
    ldfd    f72=[r33],64  // load y[8*i]
    nop.i   0
 }{ .mmi
    ldfd    f37=[r20],64  // load x[8*i+1]
    ldfd    f77=[r2 ],64  // load y[8*i+1]
    nop.i   0 ;;
 }
            ...
 {  .mmi
    ldfd    f62=[r25],64  // load x[8*i+6]
    ldfd    f102=[r18],64 // load y[8*i+6]
    nop.i   0
 }{ .mmb
    ldfd    f67=[r26],64  // load x[8*i+7]
    ldfd    f107=[r19],64 // load y[8*i+7]
    br.ctop.sptk    .b1_2 ;;
 }
```

Stop bits have been inserted after every group of two bundles, therefore the peak performance of 4 loads per cycle (0.5 cycle per "iteration", an iteration corresponding to the access of one element of X and one element of Y) should be reachable. A variant without any stop bits was also generated and tested, leading to the same results as the one given above.

The performance figure when operands are located in L2 is given in Figures 1a and 1b. Basically, two types of phenomena can be observed:

- Three diagonals separated by 256 Bytes, along which performance goes up to 1 cycle;
- A grid type pattern (sometimes overwritten by the diagonal patterns): on the "good" points (Offset X and Offset Y being of the form 16r + 8), performance is optimal (0.5 cycles) while on the other points performance degrades up to 0.9 cycles.

Both phenomena can be attributed to bank conflicts generated by the interaction between the L2 interleaving scheme and the address streams. The table below summarizes the main bank conflicts occurring with the interleaved Load Load variant. In this table only the block of the first 4 accesses is displayed, the other ones being easily obtained from this initial pattern.

Offset Y values	Offset X values	Load X(0), Load Y(0), Load X(1), Load Y(1) L2 Bank number accessed	Performance (cycles per Load/Load)
Offset Y = 0	Offset X = 0	0 0 0 0 (quadruple conflict on bank 0)	1
Offset Y = 0	Offset X = 8	0 0 1 0 (triple conflict on bank 0)	1
Offset Y = 0	Offset X = 64	8 0 8 0 (two double conflicts on bank 0 and 8)	0.9
Offset Y = 0	Offset X = 72	8 0 9 0 (double conflict on bank 0)	0.9
Offset Y = 8	Offset X = 0	0 0 0 1 (triple conflict on bank 0)	1
Offset Y = 8	Offset X = 8	0 0 1 1 (two double conflicts on bank 0 and 1)	0.9
Offset Y = 8	Offset X = 64	8 0 8 1 (double conflict on bank 8)	0.9
Offset Y = 8	Offset X = 72	8 0 9 1 (no conflict!!)	0.5

The 256 Bytes period of the diagonal patterns stems from the fact bank allocation is periodic with a 256 Bytes period: i.e. two elements of the same array, which are 256 Bytes apart, are allocated to the same cache bank.

As a main conclusion, the classic interleaved access is generating numerous bank conflicts leading to very unstable performance.

5.1.2. Vector Optimized Load/Load

The basic idea is to build a memory access pattern, such that no bank conflict occurs. Therefore, the technique is first to vectorize memory access on X and Y (grouping together 8 consecutive accesses on X then 8 consecutive accesses on Y) and then sorting them in an odd-even manner. The corresponding assembly code is given below:

```
.b1_2:
 {   .mmi
   ldfd    f32=[r32],64   // load x[8*i]
   ldfd    f37=[r21],64   // load x[8*i+2]
   nop.i   0
 }
 {   .mmi
   ldfd    f52=[r23],64   // load x[8*i+4]
   ldfd    f57=[r25],64   // load x[8*i+6]
   nop.i   0 ;;
 }
 {   .mmi
   ldfd    f42=[r20],64   // load x[8*i+1]
   ldfd    f47=[r22],64   // load x[8*i+3]
   nop.i   0
 }
 {   .mmi
   ldfd    f62=[r24],64  // load x[8*i+5]
   ldfd    f67=[r26],64  // load x[8*i+7]
   nop.i   0 ;;
 }
   ...
Loads on y[0], y[2], y[4], y[6], y[1], y[3]

{   .mmb
   ldfd    f102=[r17],64 // load y[8*i+5]
   ldfd    f107=[r19],64 // load y[8*i+7]
   br.ctop.sptk    .b1_2 ;;
 }
```

With such a scheduling of loads, every group of four loads (delimited by stop bits) necessarily address four distinct banks. The use of stop bits is important to preserve this specific grouping.

The resulting performance numbers are given on Figure 1c and 1d, the performance is optimal, (i.e. 0.5 cycle for a Load/Load pair) and flat: all of the bank conflicts have been eliminated.

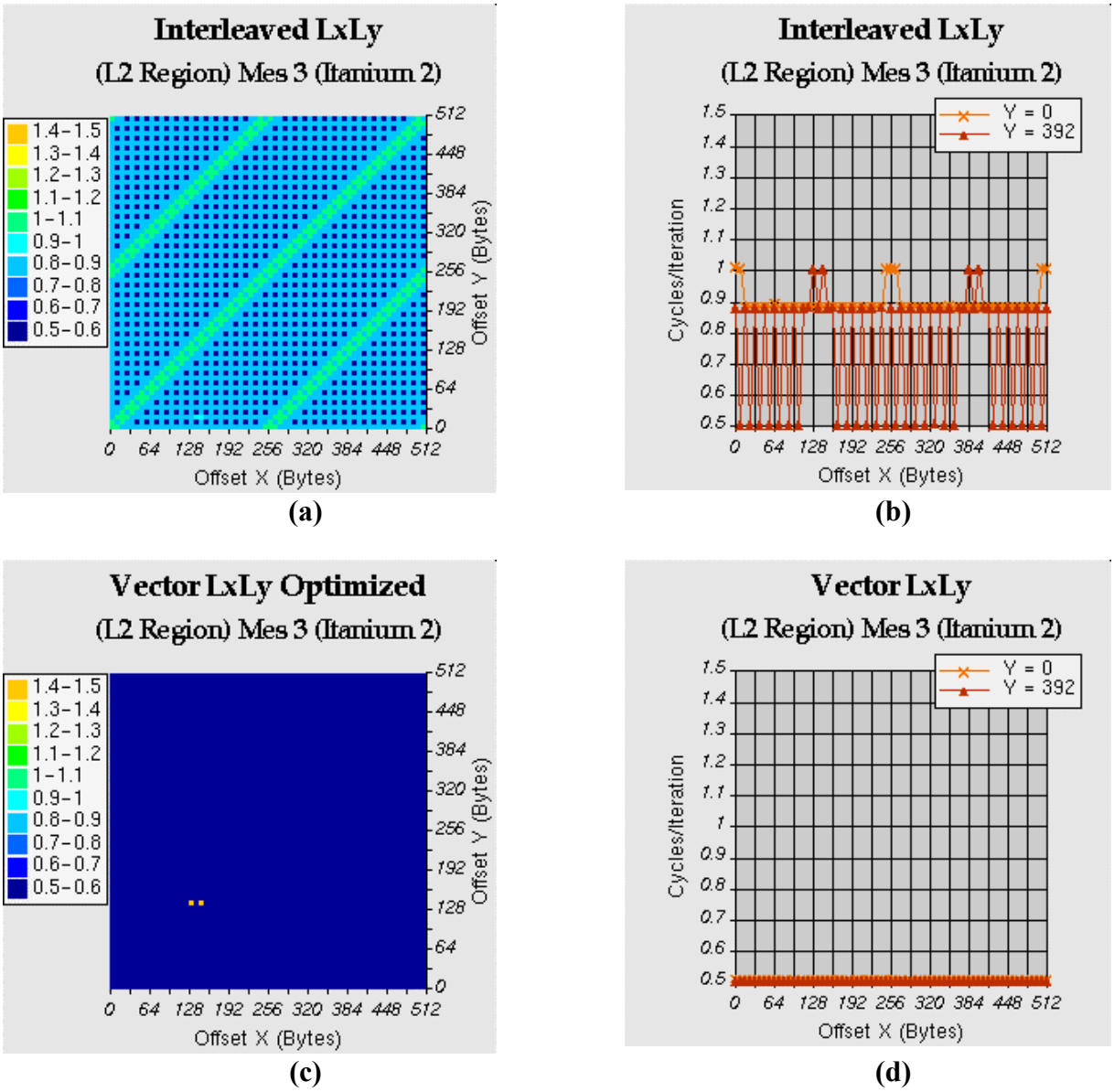


Figure 1. Interleaved LxLy and Vector LxLy Optimized (Itanium 2™).

5.2 Load Store results (Figure 2)

Again two variants have been tested: interleaved and another variant called Vector Optimized.

5.2.1 Load Store Interleaved results

The Load/Store interleaved memory access pattern is very similar to the Load/Load interleaved, the second Load on Y being replaced by a Store.

The corresponding performance results are displayed in Figure 2a and 2b. Although they present some similarities with the Load/Load interleaved variant, there are a number of differences.

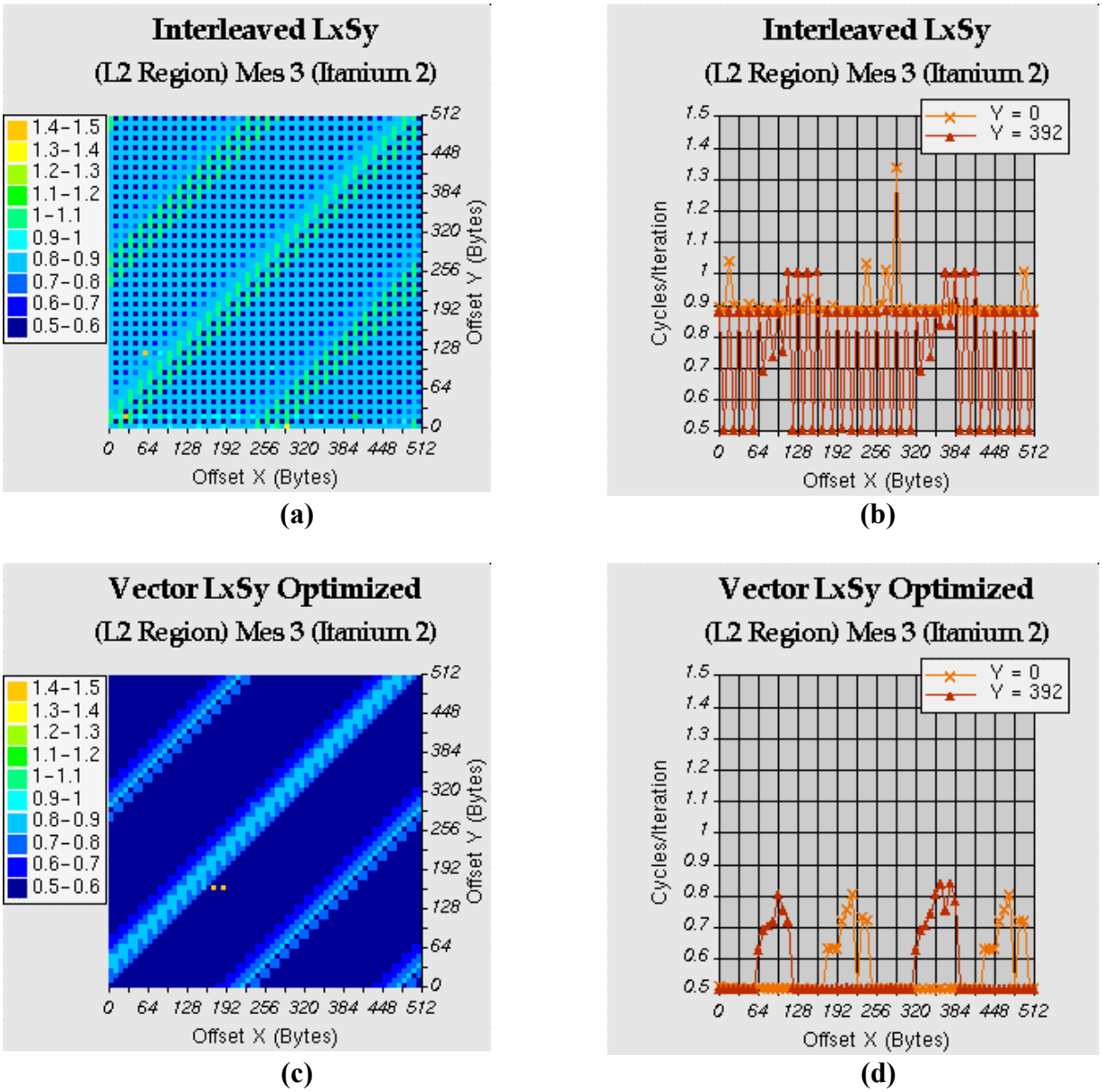


Figure 2. Interleaved LxSy and Vector LxSy Optimized (Itanium 2™).

- A grid pattern can be clearly observed. Again, it is due to bank conflicts but it is more complex to analyse than the previous because a Store will only conflict at the bank with Stores that are issued three cycles later.
- Three diagonal patterns can still be observed but the main diagonal one is higher around 1.3 cycles. This is due to the interaction between a Load and a Store, which have exactly the same low order 16 bits, therefore causing a disambiguation problem because they are issued in the same cycle.

5.2.2. Vector Optimized results

The strategy used for the Load/Load can no longer be applied here because grouping stores by blocks of 4 stores per cycle hits a key performance barrier: a maximum of two stores can be issued per cycle.

Therefore the technique used is group Load and Stores in the following manner:

Load X(0), Load X(2), Store Y(0), Store Y(2) ;;

Load X(1), Load X(3), Store Y(1), Store Y(3) ;;
and so on ….

This scheduling technique will not completely eliminate all of the pathological behaviour. It will suppress the grid pattern and part of the Load/Store disambiguation problem. However, the "bad" performance zones will be clearly defined and in fact, they could be easily derived from the memory access pattern: they will correspond to narrow "diagonal" zones. This is confirmed by the experimental results in Figures 2c and 2d. On these figures, it can be observed that performance is optimal (0.5 cycles per Load and Store, i.e. 2 Loads and 2 Stores are executed every cycle) except on diagonals zones 256B apart. The grid pattern has disappeared. To fully get rid of the diagonal zones, an additional technique based on software pipelining will have to be used. This will be detailed in the Copy kernel section. In essence, software pipelining will allow to move the diagonal and by combining two versions, all of the conflicts can be eliminate

6. Copy kernel results

To allow a fair comparison, the same scale is used in the L2 (resp. L3 region) in Figures 3a, 3b, 5a and 5b (resp. 3c, 3d, 5c and 5d).

6.1. Copy Std results (Figure 3)

For this kernel, the Compiler V6.0 was used because the V7.0 compiler recognized the Copy pattern and generated a call to a library, which turned out to be slower than the code generated by the V6.0.

The assembly code generated by the V6.0 compiler is the following:

```
.b1_3:
 {    .mmi
  // update prefetch distance
 (p16) add r32=16,r34
  // prefetch on X and Y arrays
alternately
 (p16) lfetch.nt1       [r34]
       nop.i   0
 }
 {    .mmb
  // store y[i-6]
(p22) stfd    [r3]=f38,8
  // load  x[i]
(p16) ldfd    f32=[r2],8
      br.ctop.sptk    .b1_3 ;;
 }
```

The code is far from being optimal: in particular, too many prefetch instructions are generated, one every iteration alternating between X and Y arrays. Still using such a code structure, performance levels are expected to be 1 cycle per iteration (assuming operands are located in L2).

Unfortunately, as displayed on Figures 3a and 3b, performance is at best of 1.3 cycles per iteration. Furthermore several diagonals patterns appear with performance varying between 1.6 and 2 cycles. Theses diagonal patterns are partly due to bank conflict but also to the lack of precise disambiguation (Load/Store queue problem).

The situation in L3 (Figures 3c and 3d) is very similar to the L2: best performance is around 1.7 cycles while performance along diagonal patterns varies between 2 and 3 cycles. It should be noted the presence of a wider main diagonal zone due to the interaction between Loads and Stores which are not correctly disambiguated. This phenomenon is more visible in the L3 region than in the L2 region. Another interesting phenomenon is related to the wide diagonal zone located in the upper left corner: the performance degradation occurring in this area is due to the interaction between the prefetch instruction (which behaves like a Load) and the Store instruction.

6.2. Copy Optimized results (Figure 4 and 5)

Our Copy optimized code contains two variants (variant1 and variant2), which are called depending upon the value of offset X and Y.

The variant1 is built along the same principles as the Load/Store Vector kernel. First the loop is unrolled 8 times and then the loop is software pipelined with a depth 6: load X(I) is performed at the same as Store Y(I-6). The value of 6 was chosen to accommodate the L2 Load latency. In L2 region, the performance is given in Figure 4a and 4b. As expected, performance is flat and close to optimal (around 0.6 cycles) except on a few diagonals, which are 256 Bytes apart..

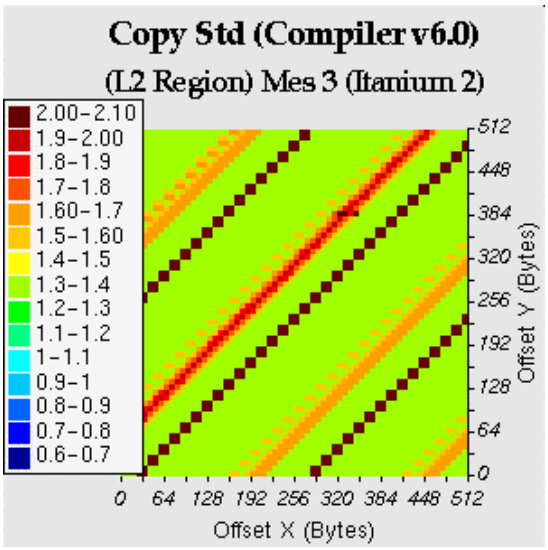
The discrepancy between 0.6 and 0.5 on the Load/Store kernel is due to the presence of prefetch instruction.

Generating variant2 is straightforward, software pipelining is pushed further i.e. at a degree of 22 (6 + 16) : Load X(I) is performed at the same time as Store Y(I-22). Such an increase in software pipeline degree is in fact equivalent to add 128 Bytes (16 x 8) to the offset X value. Now the "bad diagonal zones" have been shifted by 128 Bytes (cf. figures 4c and 4d).

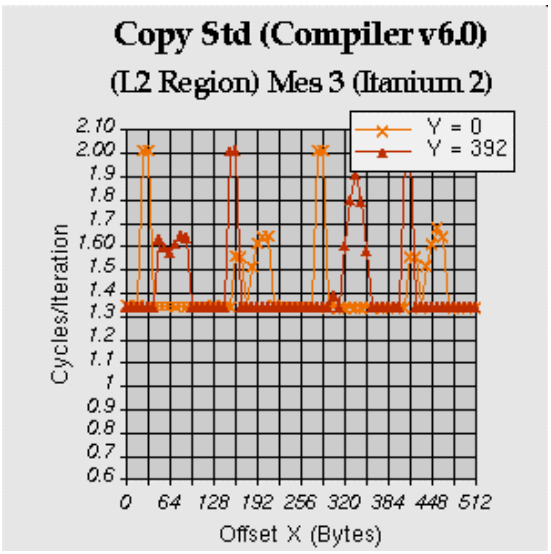
Now, it is easy to combine both variants in the same code, inserting a switch, which will choose the right variant depending upon offset X and Y values. This

generates our final code whose performance is perfectly flat (cf. Figures 5a and 5b).

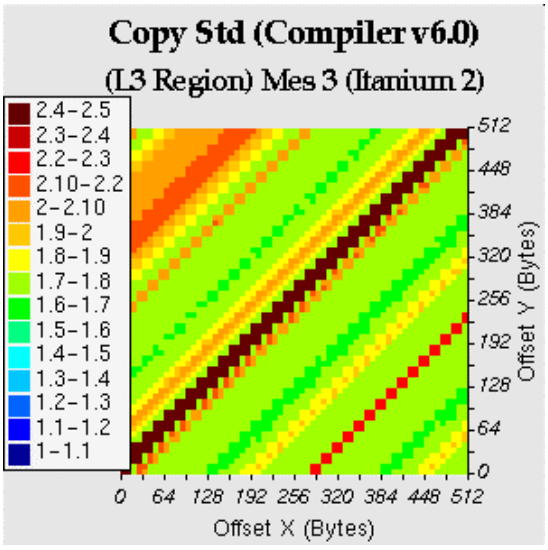
In L3 region (cf. Figures 5c and 5d) , the performance of our variant is flat and better than the compiled V6.0.



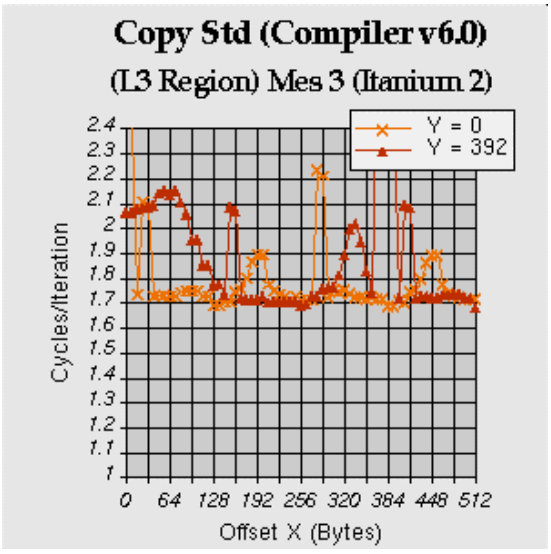
(a)



(b)

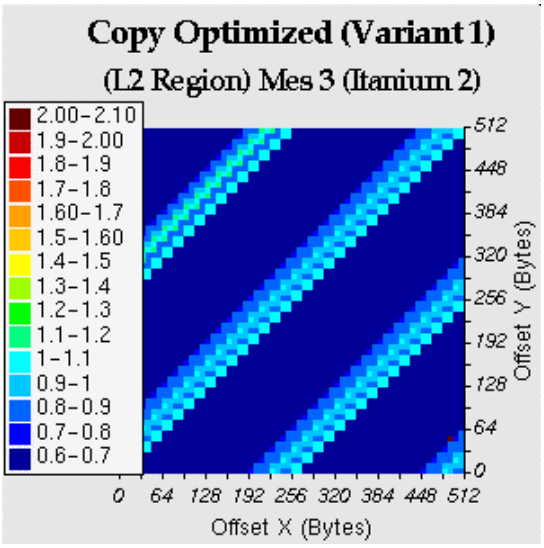


(c)

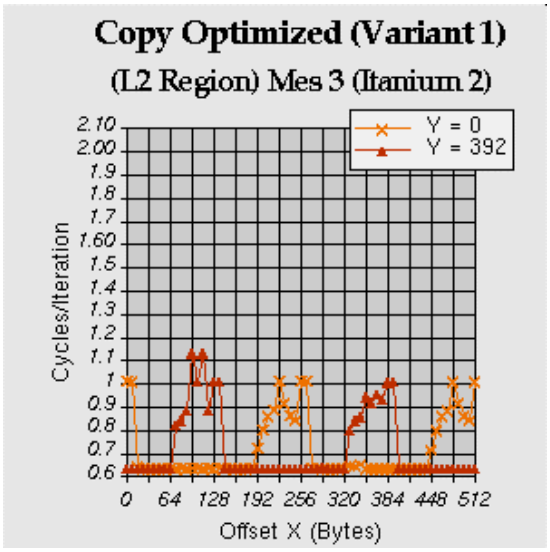


(d)

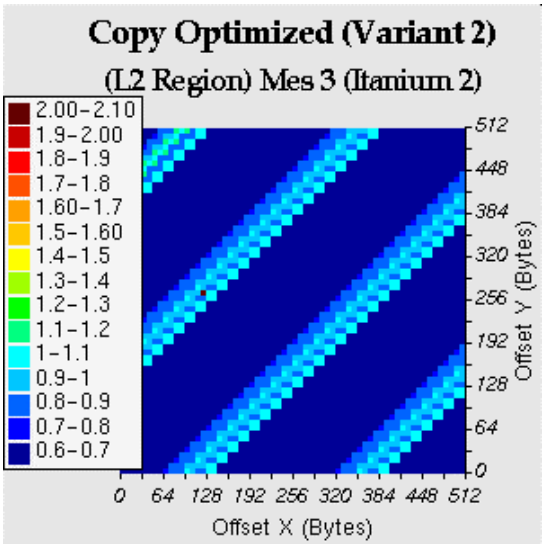
Figure 3. Copy Std (Compiler V6.0) (Itanium2™).



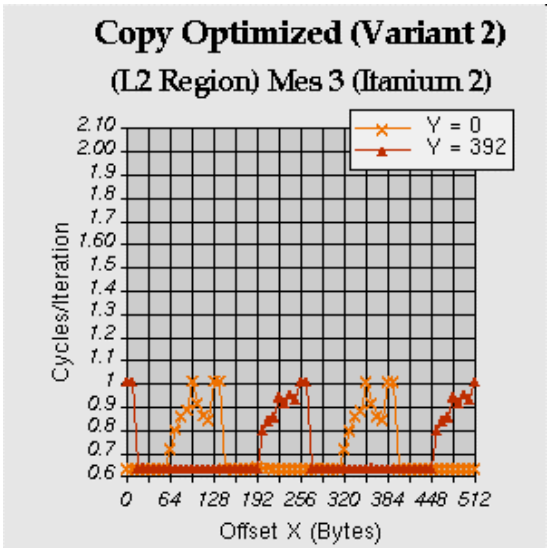
(a)



(b)

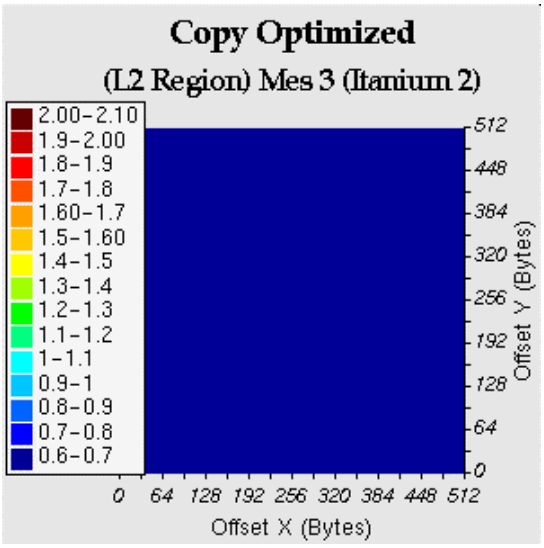


(c)

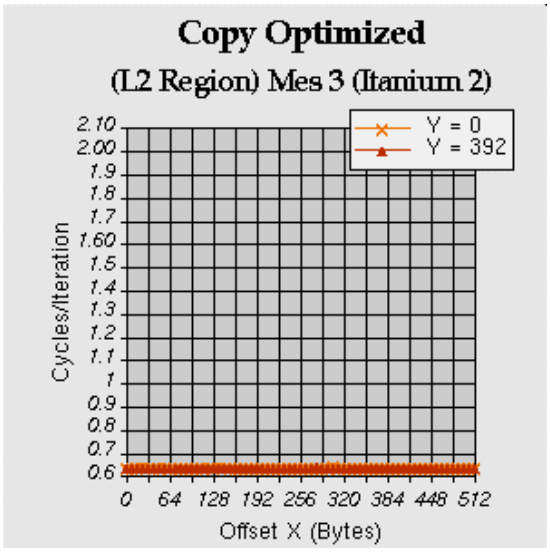


(d)

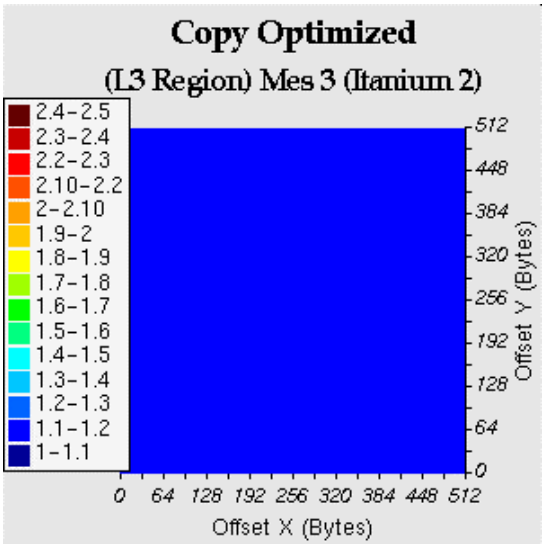
Figure 4. Copy Optimized (Variants) (Itanium2™).



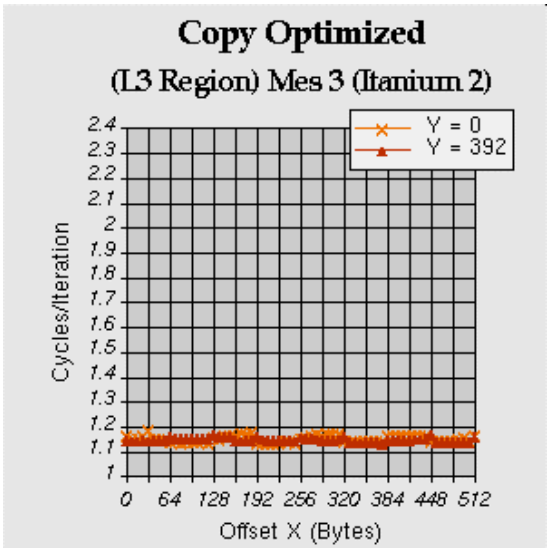
(a)



(b)



(c)



(d)

Figure 5. Copy Optimized (Itanium2™).

7. Daxpy kernel results

To allow a fair comparison, the same scale is used in the L2 (resp. L3 region) in Figures 6a, 6b, 7a and 7b (resp. 6c, 6d, 7c and 7d).

7.1. Daxpy Std results (Figure 6)

The code generated by the V7.0 compiler is fairly complex. It contains three variants, which are used depending upon loop length and Offset y values:

- Variant 1 corresponds to a code unrolled 8 times and uses Load Floating Point pair instructions on Y. Therefore this variant is only used when offset Y is a multiple of 16 Bytes;
- Variant 2 corresponds to a code also unrolled 8 times but without the use of Load Floating Pair

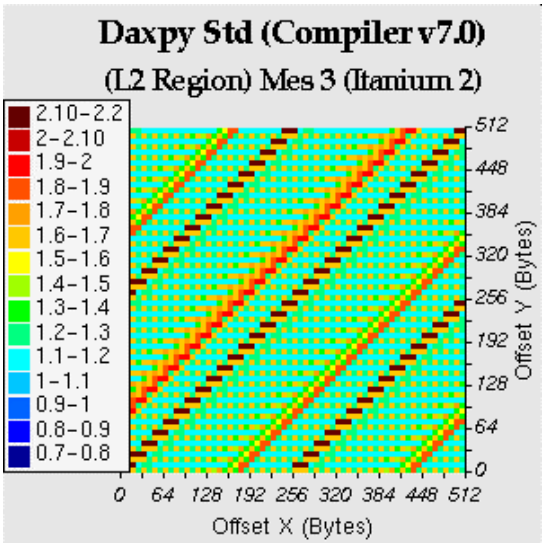
instructions. Therefore it is used when offset Y is not a multiple of 16 Bytes;

- Variant 3 is not unrolled and seems to be used for loop count.

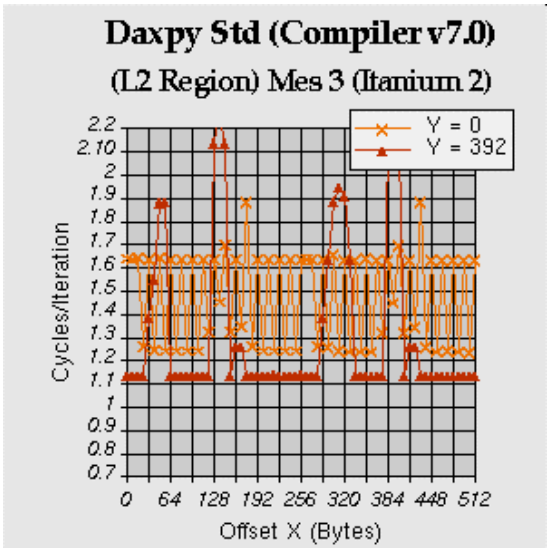
For the three variants, prefetch instructions on X and Y are inserted.

In L2 region (Figures 6a and 6b), performance is oscillating a lot. First a grid pattern can be observed somewhat similar to the one observed on Load Load Interleaved and Load Store Interleaved kernels. A detailed analysis of the code reveals that this grid pattern is clearly generated by bank conflicts: reference to arrays X and Y coexist in the same bundle. Again most of the diagonals patterns could also be attributed to bank conflicts.

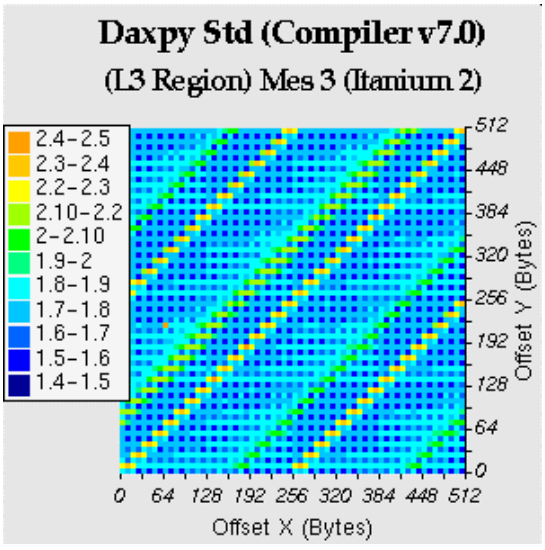
In L3 region (Figures 6c and 6d), the behavior is very similar to the one observed in L2 region. The relative amplitude of the oscillations is slightly reduced.



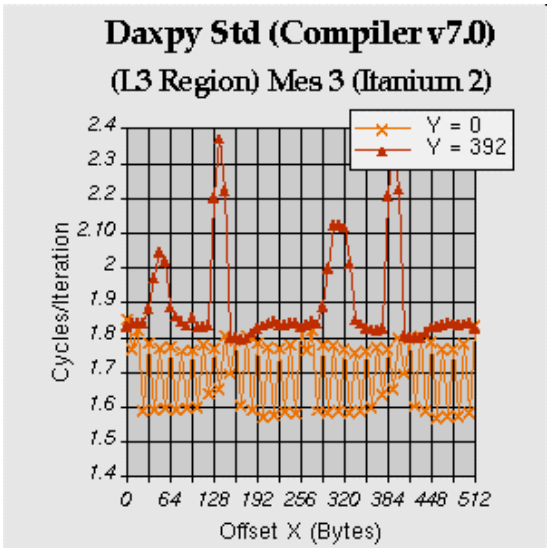
(a)



(b)



(c)



(d)

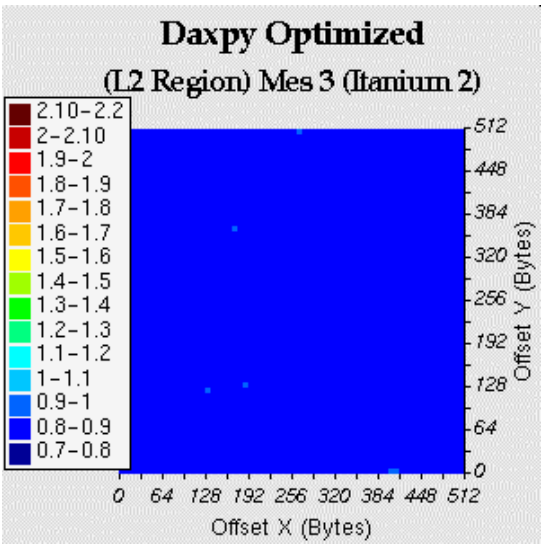
Figure 6. Daxpy Std (Compiler V7.0) (Itanium2™).

7.2. Daxpy Optimized results (Figure 7)

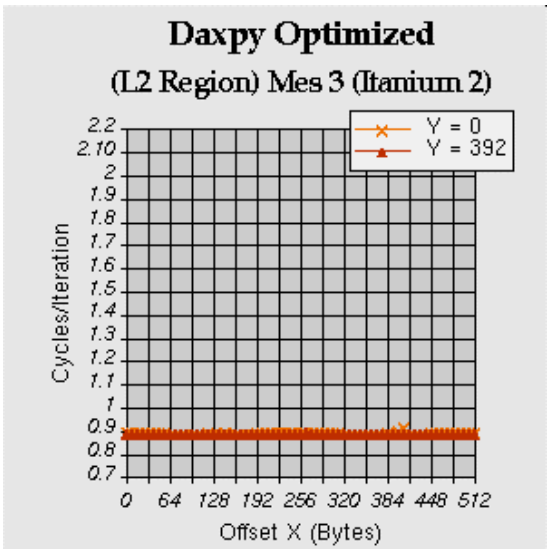
The Daxpy optimized code is obtained by combing the copy memory access pattern (Load on X and Store on Y) and the Load/Load access pattern (Load on Y).
As for the Copy, two variants are necessary to get rid of all of the diagonal zones.

The performance results are presented in Figures 7a and 7b (L2 region) and in Figures 7c and 7d (L3 region). In both cases performance is always better than the V7 compiler.

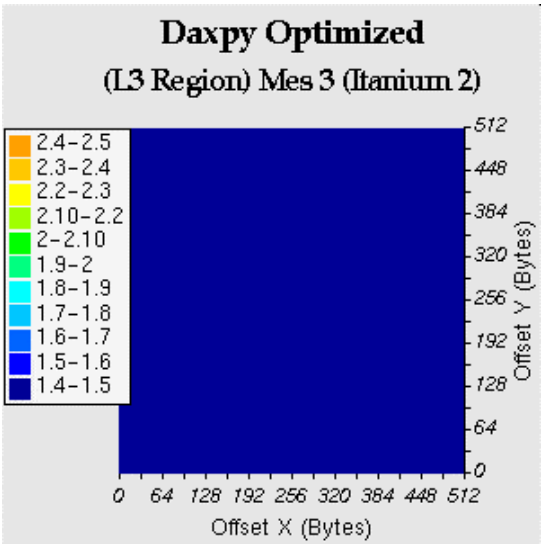
The performance of 0.9 cycles per iteration might seem disappointing while a simple count would lead to 0.75 cycles because one iteration requires two loads and one store (each of them costing 0.25 cycles). However taking into account necessary prefetch instructions, performance at best is 0.82 cycles. Now our optimized version was only unrolled 8 times and therefore, unnecessary prefetch instructions were inserted. Unrolling 16 times would lead to the optimal performance.



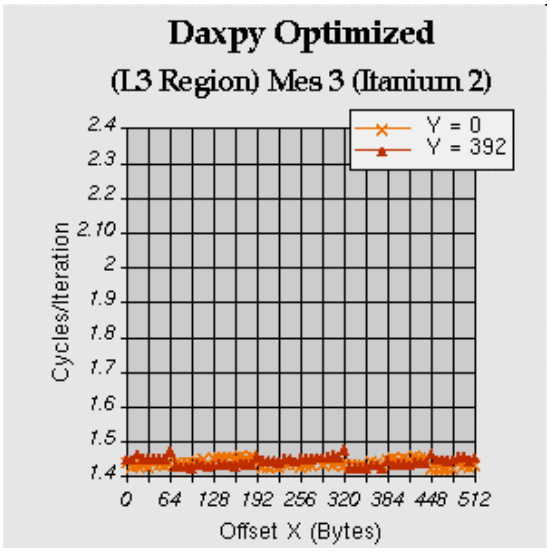
(a)



(b)



(c)



(d)

Figure 7. Daxpy Optimized (Itanium2™).

8. Conclusion

Modern microprocessors rely on complex cache systems to deliver top performance. The dark side of the coin is the resulting complexity, not only for designing them but also for exploiting them efficiently even on simple codes.

Our studies of simple BLAS1 kernels on the Itanium2™ have clearly shown the complex performance behavior of the cache system (both L2 and L3). It was clearly demonstrated that the banking structure of the L2 has a major impact on performance. This banking structure has to be taken into account when scheduling memory access instructions. We explored two specific memory access patterns (Load/Load) and (Load/Store) from which we derived efficient instruction scheduling techniques (variants around vectorization of memory access). These techniques were then applied to two standard BLAS1 primitives and demonstrated stable and close to optimal performance in particular when operands are located in L2.

This work will be extended into four major directions:

- The instruction scheduling techniques described here should be studied in more depth. Vectorization and increasing software pipeline depth increases a lot register pressure. Our Daxpy Optimized variant is close to using 80 floating-point registers!! Therefore, register consumption should be studied in more detailed manner and in particular when dealing with more complex loop bodies, involving for example a larger number of arrays.
- Main memory access deserves a similar study. Already, some preliminary tests have confirmed us with the good performance capabilities of the vectorization strategy;
- New generation of Itanium™ processors are coming up (Madison), which deserve similar studies;

Finally, other processor architectures (Power, UltraSparc, …) should be studied in a similar manner. We already performed a similar set of experiments on Alpha 21264 and Power4: similar phenomena not related directly to banking but to insufficient Load/Store queue disambiguation mechanisms were encountered. Other specific instruction scheduling techniques have to be developed to overcome these weaknesses.

Acknowledgements

This study has been funded by the French Ministry of Research and by Bull. We would like to thank Gerard Roucairol Bull CTO who has triggered our interest for IA64, Jean Papadopoulo Senior Architect at Bull who has strongly supported our effort and finally Bull's IA64 team who has provided us with a timely access to an Itanium2™ platform. We also thank Sid Touati for his careful proofreading of the manuscript.

Bibliography

[Ba95] "Unfavorable Strides in Cache Memory Systems", D.H. Bailey, Scientific Programming, vol. 4 (1995), pp. 53-58

[Ba00] "The Intel IA-64 Compiler Code Generator", Jay Bharadwaj et al, IEEE Micro, Sept -Oct. 2000.

[CB95] "Effective hardware-based data prefetching for high-performance processors", Tien-Fu Chen and Jean-Loup Baer, IEEE transactions on computers, May 1995, 44, 5, pp609-623

[CS98] "Parallel Computer Architecture a Hardware/ Software approach", David~E. Culler, Jaswinder~Pal Singh, and Anoop Gupta. Morgan Kaufman, 1998.

[FJ95] "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors", Keith I. Farkas, Norman P. Jouppi, Paul Chow, Proceedings of the International Conference on High Performance Computing Architecture, 1995

[Ha00] "Introducing the IA-64 Architecture", Jerry Huck et al., IEEE Micro, Sept - Oct. 2000.

[IN01] "Intel ® Itanium ™ Processor Reference Manual for Software Optimization", Intel Document Number: 245473-003, Nov 2001.

[IN02] "Intel ® Itanium2 ™ Processor Reference Manual for Software Development Optimization", Intel Document Number: 251110-001, June 2002

[JF01] "Application Optimization for Itanium ™ - based Systems", J. Baron, J. Fier, J.P. Panziera, A. Raefsky, Intel Development Forum, Aug 2001.

[OL85] "On the Effective Bandwidth of Interleaved Memories in Vector Systems," W. Oed and O. Lange, IEEE Transactions on Computers, C-34(10):949--957, October 1985.

[SA00] "Itanium Processor Microarchitecuture", Harsh Sharangpani, Ken Arora, IEEE Micro, Sept - Oct. 2000.

Compile-Time Cache Hint Generation for EPIC Architectures

Kristof Beyls
Ghent University,
Sint-Pietersnieuwstraat 41,
Gent, Belgium
kristof.beyls@elis.rug.ac.be

Erik H. D'Hollander
Ghent University,
Sint-Pietersnieuwstraat 41,
Gent, Belgium
erik.dhollander@elis.rug.ac.be

ABSTRACT

One of the new possibilities offered by EPIC architectures is to allow the compiler to direct the cache placement and replacement policy through cache hints. In this work, a method to generate these cache hints at compile time is presented. The generation of appropriate cache hints is based on the locality of the instructions they apply to, which is quantified by the reuse distance metric. Next to the static selection of the most appropriate cache hints, a dynamic selection of cache hints by predicates is proposed. The generation of static hints is based on a simple profiling scheme, while the dynamic selection is based on an analytical model of the programs cache behavior.

The implementation of the static approach in the Open64-compiler shows a speedup of 7% on average on a set of pointer-intensive and regular loop-based programs. The dynamic approach shows that for loop kernels, up to 35% reduction in cache misses can be attained. Furthermore, the dynamic selection allows to remove almost twice as many cache misses as statically selected cache hints.

1. INTRODUCTION

It is well-known that data cache misses form a severe bottleneck in executing programs. Typically, Itanium programs spent about 50% of their execution time on resolving cache misses[3]. In the future, the speed gap between processors and memory will continue to grow, making cache misses an even larger bottleneck. Therefore, improving the cache behavior is essential to obtain good execution speeds. In the past, many compiler optimizations have been proposed to enhance the data cache behavior. However, in traditional processors, the hardware decides when and where data is placed and replaced in the cache hierarchy, and the compiler can only influence the cache behavior indirectly. With the advent of cache hints in EPIC architectures, for the first time, the compiler has the means to steer the cache behavior directly. The challenge is to decide in the compiler which cache hints to generate.

Cache hints are annotated to memory instructions, such as loads, stores and prefetches. They have two purposes: inform the compiler about the true latency of load and prefetch operations and inform the processor at which cache level data should be placed. Cache hints are further discussed in section 2.

In order to generate appropriate cache hints, it is required that the compiler has an idea about the locality of the instructions. In this paper, the locality is quantified by the reuse distance metric. It allows to accurately determine which memory instructions result in cache misses and whether the program will profit from retaining data at a given cache level. Section 3 further discusses the reuse distance and its properties as a locality metric.

In section 4, the selection of cache hints based on the reuse distance is presented. The cache hints are an integral part of the memory instruction. Therefore, all executions of the same instruction share the same cache hint. However, the different executions of the instruction can exhibit different amounts of locality, requiring different cache hints. The presented approach selects a single cache hint which is appropriate for all executions of the instruction.

The restriction that all executions of the same memory instructions must share the same cache hint is worked around in section 5. There, a program analysis and transformation is presented to make sure that all executions of the same instruction require the same cache hint. The program analysis calculates the exact reuse distances of the different executions of a memory instruction. After the analysis, the instruction is duplicated with different cache hints and predicates are used to dynamically select the version with the most appropriate hint.

In section 6, the implementation and the experimental evaluation of the cache hint selection scheme is presented. In section 7, related work is discussed, and in section 8, the conclusion follows.

2. SOFTWARE-CONTROLLED CACHING IN EPIC

One of the basic principles of the EPIC-philosophy is to let the compiler decide when to issue operations and which resources to use. This contrasts with the superscalar paradigm, where the processor is responsible for deciding e.g. which instructions to execute in parallel, how to predict branches and where to place data in the cache hierarchy. In an EPIC architecture, the responsibility for making these choices is mostly shifted to the compiler. In order to communicate the compiler decisions to the processor hardware, they must be representable in the instruction set architecture.

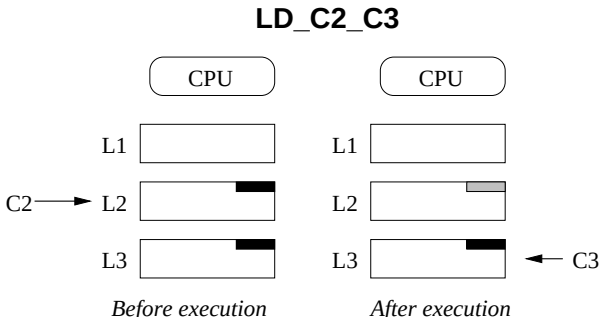


Figure 1: Example of the effect of the cache hints in the load instruction `LD_C2_C3`. The source cache specifier `C2` in the instruction suggests that the data resides in the L2-cache. The target cache specifier `C3` indicates that the data should be stored no closer than the L3-cache. As a consequence, the data is the first candidate for replacement in the L2-cache.

The existing EPIC architectures (HPL-PD[14] and IA-64[11]) communicate the compiler decisions about the cache hierarchy management to the processor through cache hints. The semantics of cache hints on both architectures are similar. First, the cache hints in HPL-PD are presented, after which their counterparts in the IA-64 architecture are discussed.

In the HPL-PD architecture, cache hints are attachments to regular memory instructions, and occur in two kinds: the source and target hints. The first kind, the *source cache specifier*, indicates at which cache level the accessed data is likely to be found. The second kind, the *target cache specifier*, indicates at which cache level the data is kept after the instruction is executed. An example is given in fig. 1, where the effect of the load instruction `LD_C2_C3` is shown.

The *source cache specifiers* are used by the compiler to know the estimated data access latency. Without these specifiers, the compiler assumes that all memory instructions hit in the L1 cache. Using the source cache specifier, the compiler is able determine the true memory latency of instructions. It uses this information to schedule the instructions explicitly in parallel. The *target cache specifiers* are used by the processor, where they indicate the highest cache level at which the data should be kept. A carefully selected target specifier will maintain the data at a fast cache level, while minimizing the cache pollution.

Similar cache hints are available in the IA-64 architecture. Since source cache hints are used inside the compiler, there's no need to communicate them to the processor. Therefore, IA-64 only defines the target cache hints `.t1`, `.nt1`, `.nt2` and `.nta`. `.t1` means that the memory instruction has temporal locality in all the cache levels. `.nt1` only has temporal locality in the L2 cache and below, but there might still be spatial locality in the L1 cache. Similarly, `.nt2` respectively `.nta` indicates that there's only temporal locality in L3 respectively no temporal locality at all.

3. REUSE DISTANCE

The reuse distance is the locality metric used in this work. It is defined within the framework of the following definitions.

Definition 1. *A* **memory reference** *corresponds to a read or write instruction, while a particular execution of that read or write at runtime is a* **memory access***[8].*

Definition 2. *A* **reuse pair** $\langle a_1, a_2 \rangle$ *is a pair of memory accesses in a memory access stream, which touch the same memory location, without intermediate accesses to that location. The* **accessed data set (ADS)** *of a reuse pair* $\langle a_1, a_2 \rangle$ *is the set of unique memory locations accessed between* a_1 *and* a_2*, and is denoted by* $\mathsf{ADS}\langle a_1, a_2 \rangle$*. The* **reuse distance** *of a reuse pair* $\langle a_1, a_2 \rangle$ *is the number of unique memory locations accessed between accesses* a_1 *and* a_2*. It is denoted by* $\mathsf{RD}(\langle a_1, a_2 \rangle)$*, and equals* $|\mathsf{ADS}\langle a_1, a_2 \rangle|$*.*

Definition 3. *Consider the reuse pairs* $\langle a_1, a_2 \rangle$ *and* $\langle a_2, a_3 \rangle$*. The* **forward reuse distance** *of a memory access* a_2 *is the reuse distance of the pair* $\langle a_2, a_3 \rangle$*. If there is no such reuse pair, its forward reuse distance is* ∞*. The* **backward reuse distance** *of* a_2 *is the reuse distance of* $\langle a_1, a_2 \rangle$*. If there is no such pair, the backward reuse distance is* ∞*. The forward reuse distance of* a_2 *is denoted by* $\mathsf{FRD}(a_2)$*, its backward reuse distance is denoted by* $\mathsf{BRD}(a_2)$*.*

Figure 2 shows three reuse pairs in a short memory access stream.

Reuse Distance Theorem. *In a fully associative LRU cache with* n *lines, an access with backward reuse distance* $d < n$ *will hit. An access with backward reuse distance* $d \geq n$ *will miss.*

In a fully associative LRU cache with n *lines, the memory line accessed by a reference with forward reuse distance* $d < n$ *will stay in the cache until the next access of that memory line. A reference with forward reuse distance* $d \geq n$ *will be removed from the cache before the next access.*

Proof. In a fully associative LRU cache with n cache lines, the n most recently referenced memory lines are retained. When a reference has a backward reuse distance d, exactly d different memory lines were referenced previously. If $d \geq n$, the referenced memory line is not one of the n most recently referenced lines, and consequently will not be found in the cache.

If the forward reuse distance is infinite, the data will not be used in the future, so there is no next access. If the forward reuse distance is not infinite, consider the forward reuse distance of access a_1 and assume that the next access to the data occurs at access a_2, resulting in a reuse pair $\langle a_1, a_2 \rangle$. By definition, the forward reuse distance d of a_1 equals the backward reuse distance of a_2, i.e. d. Therefore, the data will be found in the cache at access a_2, if and only if $d < n$. □

The theorem above indicates that the reuse distance can be used to precisely indicate the cache behavior of fully associative caches. However, previous research[10, 2, 4] indicates that also for lower-associative, and even for direct mapped caches, the reuse distance can be used to obtain a good estimation of the cache behavior. [10, 2, 4] independently measure the error made by the theorem above, when predicting cache behavior for low-associative caches. Both

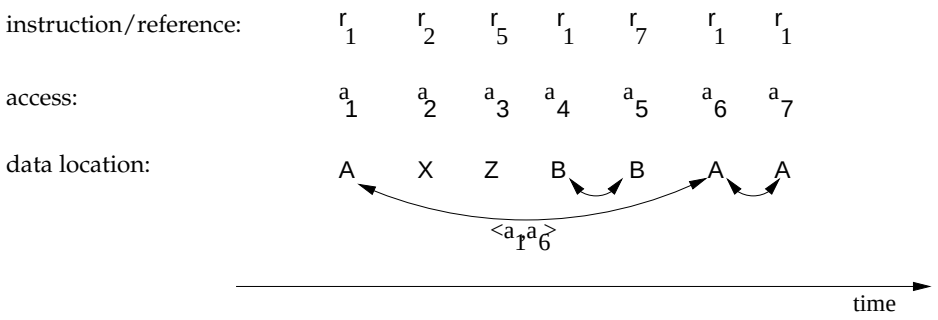


Figure 2: The top row indicates 7 memory reference instructions, generating a numbered memory access stream in the second row. The bottom row shows the corresponding memory locations A, B, X or Z. The accesses to X and Z are not part of a reuse pair, since they are accessed only once in the stream. $\mathsf{ADS}\langle a_1, a_6\rangle = \{B, X, Z\}$**, and** $\mathsf{RD}(\langle a_1, a_6\rangle) = |\mathsf{ADS}\langle a_1, a_6\rangle| = 3$**.** $\mathsf{RD}(\langle a_6, a_7\rangle) = 0$**.** $\mathsf{FRD}(a_1) = 3$**,** $\mathsf{BRD}(a_1) = \infty$**.** $\mathsf{FRD}(a_6) = 0$**,** $\mathsf{BRD}(a_6) = 3$**.**

statistical analysis and measurements based on a wide variety of program traces indicate that the relative error is low, typically less than 5%[10]. This is further confirmed by the cache behavior measurements performed on the SPEC2000 benchmark in [5]. The measurements show that for typical caches (associativity 2 or greater and cache size larger than 8KB), the capacity misses are responsible for at least 85% of all misses.

4. CACHE HINT SELECTION

In order to reduce the number of cache misses, the reuse distance of the memory accesses can be reduced so that they are smaller than the cache size, as is done by program transformations such as loop tiling[1]. However, due to dependences in the program, it is not always possible to reduce the reuse distance to be smaller than the cache size. In this paper, we exploit the possibility to adapt the instruction scheduling and the cache replacement policy through cache hints when the reuse distance is larger than the cache size.

The cache hint selection is based on the reuse distance, which predicts fully associative cache behavior perfectly. For lower-associative caches, an exact simulation of the cache could be used to know whether data is found or retained at a given cache level. However this has two disadvantages when compared to reuse distance-based selection:

1. The cache behavior which is measured for a set-associative cache is dependent on the exact layout of data in the address space. The data layout and the data alignment can change between different executions of the program. Therefore, the optimal cache hints for the measured execution are not necessarily the optimal cache hints for other executions of the program. Since the reuse distance is independent of data layout, the metric doesn't change when the data layout changes.

2. Since cache hints indicate how data should be placed and replaced in multiple cache levels, different measurements of cache behavior for all cache levels would be needed. Furthermore, they would need to be combined to generate a single cache hint. Since the reuse distance is irrespective of cache size, a single measurement allows to easily select cache hints which take into account the different cache levels.

A single memory instruction generates multiple memory accesses when that instruction is executed multiple times (e.g. in loops). The different accesses originating from the same memory instruction can exhibit different locality, requiring different cache hints. For example, in fig. 2, the instruction r_1 generates accesses a_1, a_4, a_6 and a_7. These accesses respectively have forward reuse distances 3, 0, 0 and ∞. Therefore, for the second and third access generated by instruction r_1, the data will be retained for any cache size; while for the fourth access, the data will never be reused. It is clear that the accesses originating from the same instruction require different cache hints, even though only a single hint can be specified per instruction. As a first step, we consider which cache hint is most appropriate for every single *memory access*.

Selecting Cache Hints per Access

The selection of source and target cache hints from the backward and forward reuse distance of the access is performed as follows:

- The backward reuse distance indicates the cache size which is needed for the access to be a cache hit in a fully associative cache. As described at the end of section 3, it also indicates the cache size needed for the access to be a hit for lower-associative caches with high probability. Therefore, as *source cache hint* we select *the smallest cache level that is larger than the backward reuse distance.* If a smaller (resp. larger) cache level would be selected, the compiler would assume a smaller (resp. larger) latency than the true latency. If the assumed latency is too small, the compiler doesn't try to hide the full latency of the load. If the assumed latency is too large, the compiler will generate a suboptimal schedule, because the target register of the instructions will be kept live longer than necessary.

- The forward reuse distance indicates the cache size that is needed for the data to be retained until the next reuse in a fully associative cache. As described at the end of section 3, it also indicates the cache size needed

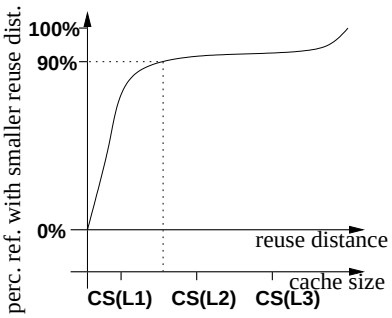


Figure 3: A cumulative reuse distance distribution for an instruction is shown and how a threshold value of 90% maps it to cache hint C2. CS(Lx) = cache size for cache level x.

to keep the data for lower-associative caches with high probability. Therefore, as *target cache hint* we select *the smallest cache level that is larger than the forward reuse distance*. In [12], it is proven that this cache hint choice per access is guaranteed to perform equal or better than the LRU replacement policy for a fully associative cache. If a smaller cache level would be selected, the data would be loaded into a small cache level where it is not retained until its reuse. Furthermore, by loading it into the small cache, that cache is polluted. If a larger cache level would be selected for the target hint, the data would not be loaded in all cache levels where it can exploit its locality.

Selecting a Single Appropriate Hint per Instruction

For every memory access, the most appropriate cache hint can be determined. However, a single memory instruction can generate multiple memory accesses during program execution. As described above, those accesses can require different cache hints. It is not possible to specify different cache hints for them, since the cache hint is specified on the instruction. As a consequence, all accesses originating from the same instruction share the same cache hint. Because of this, it is not possible to assign the most appropriate cache hint to all accesses. The following approach is used to obtain a single cache hint per instruction which is applicable for most accesses generated by the same instruction:

1. First, for every instruction, the cumulative distribution of the reuse distances of the memory accesses it generates are collected. An example of such a distribution is shown in fig. 3.
2. Based on the distribution, a source cache hint can be selected so that for at least x% of the accesses, the data will be found in that cache level. In fig. 3, it is shown how to find the cache hint so that for at least 90% of the accesses, the data will be found in the indicated cache level. Similarly, the distribution can be used to select the target cache hint so that for at least y% of the accesses, the data will be retained in that cache level.

The cumulative reuse distance distributions can be measured during a training run of an instrumented version of the program. The implementation of such a profile-based scheme in the Open64-compiler and its application to a number of benchmarks are presented in section 6.

5. DYNAMIC CACHE HINT SELECTION

When implementing the cache hint selection scheme described above, two major issues pop up. The first problem is that a cache hint is tied to an instruction, and not to a single memory access. The second problem is that the locality of the memory accesses generated by an instruction can depend on the input of the program. For example, if the program performs a matrix computation, the size of the matrices can determine the cache level where data will be found. Therefore, the optimal cache hints are also dependent on information that is in general only known at runtime. In order to mitigate the above problems, cache hints should be selected at run-time, based on the actual reuse distance of the current access. In order to be able to select the best cache hint for every single access, the reuse distance is determined analytically. Below, the analytical calculation of the reuse distance for a class of loop-oriented programs is described.

5.1 Reuse Distance Calculation

5.1.1 Program model

The analytical calculation of the reuse distance applies to programs or program parts which satisfy the following conditions: The program consists of assignment, if and loop statements. The iteration spaces of the statements must be describable by a Presburger formula[17], which represent a set of integer points that are described by a combination of linear constraints. The variables in the program are either scalars or arrays. Scalars are treated as one-dimensional arrays with size 1. Loop induction variables are assumed to reside in registers and not to generate any memory accesses. The index expressions of the array variables must be affine functions of the loop induction variables and program parameters. In order to be consistent with the default terminology in analytical calculation of cache behavior, reference is used as a synonym for instruction in this section[8]. In order to calculate the reuse pairs, their accessed data sets ADS and the corresponding reuse distance, the set of references and their iteration spaces, the set of array variables, and the order of two accesses is needed. They are formally denoted as follows:

Definition 4. *The* **set of all the references** *in a program is denoted by* $\mathcal{R}$. *The* **set of variables** *in a program is denoted by* $\mathcal{V}$. *The* **iteration space** *of the statement in which a reference r occurs is denoted by* $\mathsf{IS}(r)$. *The* **memory location** *which is accessed by r at iteration point i is denoted by* $r@i$. *The fact that iteration point i of reference r is* **executed before** *iteration point j of reference s is expressed as* $i_r \lessdot j_s$.

Example 1. *In fig. 4, a small loop is shown which can be handled by our program model. To clarify the notations introduced in definition 4, some examples applied to the loop in fig. 4 are given here:*

- *The variable set* $\mathcal{V} = \{A\}$.

```
DO i=1,N
 A(2i,1)=1
 DO j=i+2, N-i
  IF (i<>j) A(i,j-i) = A(3+j,i)
 ENDDO
ENDDO
```

Figure 4: Example loop to demonstrate the program model.

- *The reference set* $\mathcal{R} = \{\mathtt{A(2i,1)}, \mathtt{A(i,j-i)}, \mathtt{A(3+j,i)}\}$.
- *The iteration space* $\mathsf{IS}(\mathtt{A(i,j-i)}) = \{(i,j) : 1 \leq i \leq N \wedge i+2 \leq j \leq N-i \wedge \neg(i=j)\}$.
- *The mapping of iteration to data space* $\mathtt{A(3+j,i)}@(i=3, j=6) = A(9,3)$.
- *The lexicographical order constraint* $(i)_{\mathtt{A(2i,1)}} \lessdot (i',j')_{\mathtt{A(i,j-i)}} \equiv i \leq i'$.

5.1.2 Formal Reuse Equations

The reuse distances of the individual references in the program is calculated in 3 steps:

1. The reuse pairs in the memory access stream are formulated. For every couple of *references* (r,s), a single formula $\mathsf{reuse}\,(r \rightarrow s)$ is generated. The formula represents all reuse pairs for which the first access is generated by an execution of reference r, and the second access is generated by s.
2. For each set of reuse pairs, a single formula is formed which symbolically describes the accessed data set (ADS) of the reuse pairs in the set.
3. The number of different memory locations in the ADS is counted which equals the reuse distance of the reuse pair. The count is expressed by an Ehrhart polynomial[7].

The three steps are explained in further detail below. Examples of the formulas generated by the three steps can be found in section 5.1.3.

1. Reuse pair

Every memory access is uniquely defined by the reference r which generates the access, and the iteration point I_r at which the access occurs.

All reuse pairs $\langle x,y\rangle$ for which the first access x originates from reference r and the second access y originates from reference s, are combined into the set of reuse pairs denoted by $\mathsf{reuse}\,(r \rightarrow s)$, which contains the iteration points I_r and J_s that generate a reuse:

$$\forall r,s \in \mathcal{R} : \mathsf{reuse}\,(r \rightarrow s) = \tag{1a}$$
$$\big\{(I_r,J_s) : I_r \in \mathsf{IS}(r) \wedge J_s \in \mathsf{IS}(s) \wedge \tag{1b}$$
$$I_r \lessdot J_s \wedge \tag{1c}$$
$$r@I_r = s@J_s \wedge \tag{1d}$$
$$\forall t \in \mathcal{R} : \neg\,(\exists K_t \in \mathsf{IS}(t) : I_r \lessdot K_t \lessdot J_s \wedge t@K_t = r@I_r)\,\big\} \tag{1e}$$

The above formula gives the constraints which must be satisfied before a reuse occurs between $r@I_r$ and $s@J_s$. Equation (1b) expresses that I_r and J_s are part of the iteration space of respectively r and s. (1c) demands that I_r must be executed before J_s; (1d) encodes that the same memory location must be accessed; and (1e) ensures that no intervening memory access touches the same memory location. Furthermore, the following formulas define the iteration points at which *forward* respectively *backward* reuse occurs:

$$\begin{aligned}&\mathsf{reuse_F}\,(r) = \\ &\{I_r : \exists s \in \mathcal{R}, J_s \in \mathsf{IS}(s) : (I_r,J_s) \in \mathsf{reuse}\,(r \rightarrow s)\} \\ &\mathsf{reuse_B}\,(s) = \\ &\{J_s : \exists r \in \mathcal{R}, I_r \in \mathsf{IS}(r) : (I_r,J_s) \in \mathsf{reuse}\,(r \rightarrow s)\}\end{aligned} \tag{2}$$

2. Accessed data set of a reuse pair

The second step is to calculate the accessed data set (ADS) of a reuse pair. First, the function map_r is defined, which maps an iteration space to the elements of array V which are accessed by r in that iteration space, see eq. (3). Furthermore, $\mathsf{iters}_t(I_r,J_s)$ is the set of iterations of reference t which are executed between iteration I_r and iteration J_s:

$$\mathsf{map}_r = \{I \rightarrow r@I : I \in \mathsf{IS}(r)\} \tag{3}$$

$$\mathsf{iters}_t(I_r,J_s) = \{K_t \in \mathsf{IS}(t) : I_r \lessdot K_t \lessdot J_s\} \tag{4}$$

With the help of equations (3) and (4), the function $\mathsf{ADS}_V\,(\mathsf{reuse}\,(r \rightarrow s))$ is defined, which indicates the elements of array V that are in the ADS of the reuse pairs in $\mathsf{reuse}\,(r \rightarrow s)$:

$$\mathsf{ADS}_V\,(\mathsf{reuse}\,(r \rightarrow s)) = \bigcup_{t \in \mathcal{R}} \mathsf{map}_t(\mathsf{iters}_t(\mathsf{reuse}\,(r \rightarrow s))), \tag{5}$$

where $\mathsf{map}_t(\mathsf{iters}_t(\mathsf{reuse}\,(r \rightarrow s)))$ means applying function map_t to the set of iterations $\mathsf{iters}_t(\mathsf{reuse}\,(r \rightarrow s))$. Equation (5) expresses that the ADS of a reuse pair can be found by first calculating the iterations between use and reuse. Then, the ADS is simply all the data locations which are touched by the accesses in the iterations between use and reuse.

3. Reuse distance of a reuse pair

In order to find the reuse distance of a reuse pair, the number of different memory locations in its ADS needs to be counted:

$$\mathsf{RD}(\mathsf{reuse}\,(r \rightarrow s)) = \sum_{V \in \mathcal{V}} \big|\mathsf{ADS}_V\,(\mathsf{reuse}\,(r \rightarrow s))\big| \tag{6}$$

$\mathsf{ADS}_V\,(\mathsf{reuse}\,(r \rightarrow s))$ is a Presburger formula in general, which represents a set of integer points. Counting the number of elements in such a set (such as needed for $\big|\mathsf{ADS}_V\,(\mathsf{reuse}\,(r \rightarrow s))\big|$), can be performed by the methods discussed in [7] or [18]. Besides calculating the reuse distance of a reuse pair, it is also possible to compute the forward and backward reuse distances of a memory reference r. These are denoted by

$\mathsf{FRD}(r)$ and $\mathsf{BRD}(r)$:

$$\mathsf{FRD}(r) = \sum_{s \in \mathcal{R}, V \in \mathcal{V}} \left|\mathsf{ADS}_V\left(\mathsf{reuse}\left(r \rightarrow s\right)\right)\right| \quad (7)$$

$$\mathsf{BRD}(s) = \sum_{r \in \mathcal{R}, V \in \mathcal{V}} \left|\mathsf{ADS}_V\left(\mathsf{reuse}\left(r \rightarrow s\right)\right)\right| \quad (8)$$

Furthermore, the reuse distance theorem can be used to calculate at which iteration points the data will not be found in the cache and for which iteration points the data will not be retained in the cache. The iteration points where no backward reuse occurs are those where the data is first fetched, and results in a cold miss. The iteration points at which a cold miss occurs for reference r are denoted by $\mathsf{COLDM}(r)$:

$$\mathsf{COLDM}(r) = \{I : I \in \mathsf{IS}(r) \wedge I \notin \mathsf{reuse}_{\mathsf{B}}\left(r\right)\} \quad (9)$$

Similarly, the iteration points at which there is reuse, but larger than the cache size, exhibits capacity misses, denoted by $\mathsf{CAPM}(r)$:

$$\mathsf{CAPM}(r) = \{I : \mathsf{BRD}(r) \geq CS \wedge I \in \mathsf{reuse}_{\mathsf{B}}\left(r\right)\}, \quad (10)$$

where CS denotes the cache size. The iteration points at which the accessed data will not be retained in the LRU cache can be computed taking into account the reuse distance theorem, and is denoted by $\mathsf{NOKEEP}(r)$:

$$\mathsf{NOKEEP}(r) = \{I : \mathsf{FRD}(r) \geq CS \wedge I \in \mathsf{reuse}_{\mathsf{F}}\left(r\right)\}, \quad (11)$$

5.1.3 Example

As an example, the actual formulas generated during reuse distance calculation are presented for two different programs: the matrix multiplication and Cholesky factorization. The matrix multiplication is a simple loop kernel which is very well known in cache optimizations. The Cholesky factorization is a more complicated loop kernel, with multiple loop nests which are not perfectly nested.

Matrix Multiplication

The matrix multiplication code is shown in fig. 5(a). As an example, the backward reuse distance of reference `A(I,K)` at all its iteration points is calculated step by step here. The first step is to find the references that generate reuse pairs with `A(I,K)`. There are no other references in the program that can access the same data as `A(I,K)`, so only $\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right)$ needs to be calculated. Using equation (1a,...,1e), this leads to the following expression after simplification, where $I = (i, j, k)$ and $I' = (i', j', k')$:

$$\begin{aligned}&\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right) = \\ &\{(I, I') : (i, j+1, k) = (i', j', k') \text{ and } I \in \mathsf{IS}(A(I,K)) \text{ and} \\ &\quad I' \in \mathsf{IS}(A(I,K))\}\end{aligned}$$

In order to find the data that is accessed between reuses, the accesses that are executed between reuses must be calculated. For example, $\mathsf{iters}_{\mathtt{B(K,J)}}(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right))$ is calculated as follows:

$$\begin{aligned}&\mathsf{iters}_{\mathtt{B(K,J)}}(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right)) = \{(I'') : I \lessdot I'' \lessdot I' \\ &\quad \text{and } (i, j+1, k) = (i', j', k') \\ &\quad \text{and } I \in \mathsf{IS}(A(I,K)) \text{ and } I' \in \mathsf{IS}(A(I,K))\}\end{aligned}$$

```
DO I=1,N
 DO J=1,N
  DO K=1,N
   C(I,J) = C(I,J) + A(I,K)*B(K,J)
  ENDDO
 ENDDO
ENDDO
```

(a) The matrix multiplication code

```
DO J=1,N
  DO L=J,N
    DO K=1,J-1
       A(l,j) = A(l,j)-A(l,k)*A(j,k)
    ENDDO
  ENDDO
  A(j,j) = SQRT(A(j,j))
  DO M=J+1,N
    A(m,j) = A(m,j) / A(j,j)
  ENDDO
ENDDO
```

(b) Cholesky factorization

Figure 5: source codes of matrix multiply and Cholesky factorization

which expands to

$$\begin{aligned}\{(i'', j'', k'') : &(1 \leq i'' = i' \leq N \wedge 1 \leq j' - 1 = j'' \leq N \\ &\wedge 1 \leq k' \leq k'' \leq N) \\ \vee &(1 \leq i'' = i' \leq N \wedge 1 \leq j' = j'' \leq N \\ &\wedge 1 \leq k'' < k' \leq N)\}\end{aligned}$$

In the equation above, the expression $k'' < k'$ shows that it is assumed that reference `A(I,K)` occurs before `B(K,J)` in the same iteration. Now, the data accessed between reuses can be calculated, using equation (5):

$$\begin{aligned}&\mathsf{ADS}_{\mathtt{B}}\left(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right)\right) = \\ &\mathsf{map}_{\mathtt{B(K,J)}}\left(\mathsf{iters}_{\mathtt{B(K,J)}}(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right))\right) = \\ &= \{(x, y) : (k' \leq x \leq N \wedge 1 \leq y = j' - 1 \leq N) \vee \\ &\quad (1 \leq x < k' \wedge 1 \leq y = j' \leq N)\}\end{aligned}$$

The equation above shows, for every iteration point (i', j', k') of `A(i,j)` where reuse occurs, the part of array `B` that is accessed between reuses. Similarly, the data of array `A` and array `C` accessed between reuses can be calculated as follows:

$$\begin{aligned}&\mathsf{ADS}_{\mathtt{A}}\left(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right)\right) = \\ &\quad \{(x, y) : 1 \leq y \leq N \wedge 1 \leq x = i' \leq N \wedge y \neq k'\} \\ &\mathsf{ADS}_{\mathtt{C}}\left(\mathsf{reuse}\left(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}\right)\right) = \\ &\quad \{(x, y) : 1 \leq x = i' \leq N \wedge 1 \leq j' - 1 = y < N\} \\ &\qquad \cup \{(x, y) : 1 \leq x = i' \leq N \wedge 2 \leq y = j' \leq N\}\end{aligned}$$

To calculate the backward reuse distance for all the iteration points of `A(I,K)` with reuse, the number of array elements

accessed is counted, as described in equation (8):

$$\begin{aligned}\mathsf{BRD}(\mathtt{A(I,K)}) &= |\mathsf{ADS}_{\mathtt{A}}(\mathsf{reuse}(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}))| \\ &+ |\mathsf{ADS}_{\mathtt{B}}(\mathsf{reuse}(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}))| \\ &+ |\mathsf{ADS}_{\mathtt{C}}(\mathsf{reuse}(\mathtt{A(I,K)} \rightarrow \mathtt{A(I,K)}))| \\ &= N - 1 \ + \ N \ + \ 2 = 2N + 1\end{aligned}$$

The backward reuse distance of `A(I,K)` is $2N + 1$ at every iteration point with reuse. Therefore, using equations (10) and (9), the capacity and cold misses are calculated to be:

$$\begin{aligned}&\mathsf{CAPM}(\mathtt{A(I,K)}) = \\ &\{(i,j,k) : 2N + 1 \geq CS \ \wedge \ (i,j,k) \in \mathsf{reuse}_{\mathtt{B}}(\mathtt{A(I,K)})\} \\ &\mathsf{COLDM}(\mathtt{A(I,K)}) = \{(i,j,k) \in \mathsf{IS}(\mathtt{A(I,K)}) : j = 1\}\end{aligned}$$

Cholesky Factorization

The Cholesky factorization code is shown in fig. 5(b). It consists of non-perfectly nested loops. Due to space limitations, we show the calculated backward reuse distances for only one of the references. The different reuse distance domains for the iteration space of reference `A(m,j)` is shown on the left hand side of figure 6. In the middle of fig. 6, the actual backward reuse distance is shown for the different iterations. In comparison to the cumulative reuse distance distribution on the right hand side, the analytical calculation produces more exact information: for every single access, the exact reuse distance is known.

5.2 Dynamic Cache Hint Selection by Predicates

The analytical calculation allows to select the appropriate cache hints for every access at run-time. In contrast to a profile-based method, which records the reuse distance distribution, the analytical method generates a polynomial representing the reuse distance at a given iteration point. As an example, the result of a profile-driven measurement of the reuse distance is shown on the right hand side in fig. 6. For the same instruction, the analytical method generates the data represented in the table in the same figure. From the distribution, it is not possible to find out which executions of the instruction have which reuse distance. In contrast, the calculated polynomial allows to find out the reuse distance for every execution of the instruction. This extra information can be used to select the appropriate cache hint at run-time.

In order to select the most appropriate cache hint, the load instruction is duplicated with different cache hints. The instruction with the appropriate hint can then be selected using predicates. The following code is an example of this. Assume that the reuse distance value for the given iteration is calculated and stored in register `r10`. The original load instruction loads to register `r5`. `CS1` and `CS2` are the cache sizes of the first level and second level cache. The following IA-64 code executes a single load instruction with the appropriate cache hint, according to the calculated reuse distance:

```
     cmp.lt     p6, p7 = r10, CS1 ;;
(p7) cmp.ge.unc p8, p7 = r10, CS2
        // p6 = RD<CS1
        // p8 = RD>=CS2; p7 = CS1<=RD<CS2
(p6) ld.t1      r5 = ...          ;;
(p7) ld.nt1     r5 = ...
(p8) ld.nta     r5 = ...
```

6. RESULTS

6.1 Compiler Implementation

The static cache hint selection scheme presented in section 4 has been implemented in the Open64 compiler, which is based on SGI's Pro64 compiler. The reuse distance distributions for the memory instructions are obtained by instrumenting and profiling the program. After profiling, the cumulative reuse distance distribution is saved to disk. During the feedback compilation, this profile data is read in by the compiler. The source and target cache hints are annotated to the memory instruction, based on the profile data. The instruction scheduler was adapted so that it assumes the latency indicated by the source cache hint. After instruction scheduling, the compiler produces the IA-64 assembly code with target cache hints. All compilations were performed at optimization level -O2, the highest level at which instrumentation and profiling is possible in the Open64 compiler. The existing framework doesn't allow to propagate the feedback information through some optimizations phases at level -O3.

6.2 Predicated Cache Hint Selection

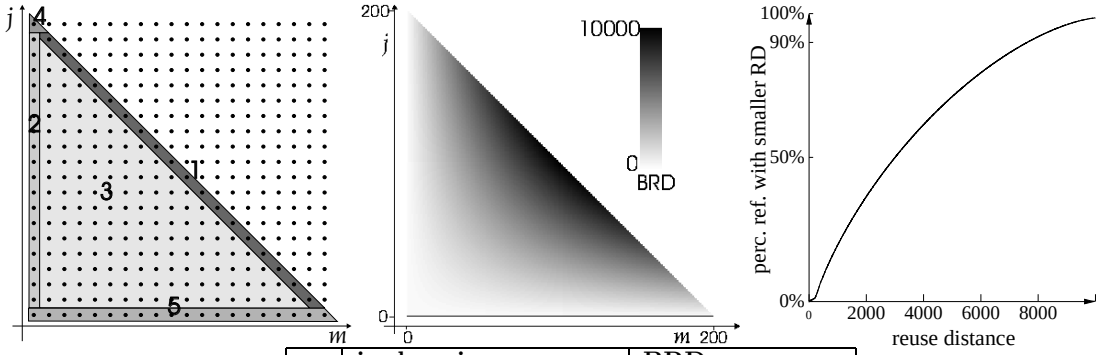
The calculation of reuse distances, as described in section 5.1, has been implemented in the FPT[26] compiler. The Omega library[18] is used to simplify the formulas and Polylib[7] is used to count the number of integer points in the sets described by the formulas.

In order to verify the exactness of the proposed equations for calculating reuse distances and cache behavior, they have been calculated automatically for a number of loop-oriented programs, such as the matrix multiplication, Gauss-Jordan elimination and Cholesky factorization. Furthermore they were applied to a number of artificial loop nests which were constructed specially to lead to far more irregular reuse distances. The results of the analysis were compared to cache simulation. The analytical results and the simulation results were identical in all cases.

6.3 Experiments

The static selection of cache hints was evaluated on a HP rx4610 multiprocessor, equipped with 733MHz Itanium processors. The data cache hierarchy consists of a 16KB L1, 96KB L2 and a 2MB L3 cache. The hardware performance counters of the processor were used to obtain detailed micro-architectural information, such as processor stall time because of memory latency and cache miss rates.

The programs were selected from the Olden and the Spec95fp benchmarks. The Olden benchmark contains programs which uses dynamic data structures, such as linked lists, trees and quadtrees. The Spec95fp programs are numerical programs with mostly regular array accesses. For the Spec95fp, the profiling was done using the train input sets, while the speedup measurements were done with the large input sets. For Olden, no separate input sets are available, and the



nr	in domain	BRD
1	$m = N \wedge 2 \leq j \leq N-2$	$m - j$
2	$3 \leq m = j + 1 < N$	$(N - m + 1)j$
3	$2 \leq j < m - 1 < N - 1$	$(N - m)j + m - 1$
4	$j + 1 = m = N \geq 3$	1
5	otherwise	∞

Figure 6: The iteration space of reference `=A(m,j)` (see fig. 5(b)) is shown on the left for $N = 20$ and is divided into 5 domains. The table shows the calculated parametric backward reuse distances for the 5 domains. In the middle, the backward reuse distance of the points in the iteration space of the same reference are shown by color, for N=200. If the pixel representing the iteration is white, the BRD is 0, when it's black, the BRD is 10000. On the right hand side, the cumulative reuse distance distribution is shown for this reference.

training input was identical to the input for measuring the speedup. The results of the measurements can be found in table 1.

The table shows that the programs run 7% faster on average, with a maximum execution time reduction of 36%. In the worst case, a slight performance degradation of 2% is observed. On average, the Olden benchmarks do not profit from the source cache specifiers. To take advantage of the source cache specifiers, the instruction scheduler must be able to find parallel instructions to fit in between a long latency load and its consuming instructions. In the pointer-based Olden benchmarks, the scheduler finds little parallel instructions, and cannot profit from its better view on the cache behavior. On the other hand, in the floating point programs, on average a 10% speedup is found because of the source cache hints. Here, the loop parallelism allows the compiler to find parallel instructions, mainly because it allows it to software pipeline the loops with long latency loads. In this way, the latency is overlapped with parallel instructions from different loop iterations. Some of the floating point programs didn't speedup a lot when employing source cache specifiers. The scheduler couldn't generate better code since the long latency of the loads demanded too many software pipeline stages to overlap it. Because of the large number of pipeline stages, not enough registers were available to actually create the software pipelining code.

The table also shows that the target cache specifiers improve both kinds of programs by the same percentage.

The method based on profiling sets cache hints per instruction. The analytical calculation gives the exact forward reuse distance for every execution of a memory instruction. Here, the additional advantage of being able to select cache hints per access instead of per instruction is quantified. IA-64-like cache hints were generated from the calculated FRD, assuming a cache line size of 1 element, so that the forward temporal locality is measured. A single cache level was simulated, which reacts similar to cache hints as the Itanium and Itanium2 processors do. When the hint indicates temporal locality, the data is placed in the cache and marked as most recently used. When the hint indicates no temporal locality, the line is still brought into the cache, to exploit potential spatial locality. However, in order not to throw out too much data with temporal locality, the line is marked as the next to be replaced.

The data cache miss rates for a number of loop-oriented programs were measured. The loop kernels and the relative miss rates for the programs compiled without cache hints, with static cache hints (per instruction) and with dynamic cache hints (per access) are shown in table 6.3.

The table shows that on average the number of misses is reduced by 6.7% by static cache hint selection and by 11.7% by dynamic cache hint selection. The program which profits most from switching from static to dynamic cache hints is Cholesky. The reason for this is that it is the program with the most irregular reuse patterns. An example of the irregular reuse distances for this program can be seen in fig. 6, which shows the different reuse distances that are generated by a single instruction. Since the same instruction requires different cache hints, static hints cannot improve the cache behavior. On the other hand, dynamic hints allow to provide every access with the most appropriate cache hint, which results in a slight cache miss reduction.

7. RELATED WORK

The speed gap between processor and memory has been doubling every two years since 1980. This observation has attracted a lot of researchers to come up with improved caching schemes. Most proposed schemes can be categorized into either hardware modifications, loop transforma-

	program	mem. stall	mem. stall reduction	source CH speedup	target CH speedup	missrate reduction L1	missrate reduction L2	missrate reduction L3	overall speedup
Olden	bh	26%	0%	0%	-1%	1%	-20%	-3%	**-1%**
	bisort	32%	0%	0%	0%	0%	6%	-5%	**0%**
	em3d	77%	25%	6%	20%	-28%	-3%	35%	**23%**
	health	80%	19%	2%	16%	0%	-1%	15%	**20%**
	mst	72%	1%	0%	0%	-10%	1%	2%	**1%**
	perimeter	53%	-1%	-1%	-1%	-11%	-56%	-6%	**-2%**
	power	15%	0%	0%	0%	-14%	2%	0%	**0%**
	treeadd	48%	0%	-2%	-1%	-2%	26%	17%	**0%**
	tsp	20%	0%	0%	0%	2%	7%	7%	**0%**
	Olden avg.	47%	5%	0%	4%	-6%	-6%	7%	**5%**
Spec95fp	swim	78%	0%	0%	1%	32%	0%	0%	**0%**
	tomcatv	69%	33%	7%	4%	-11%	-43%	6%	**9%**
	applu	49%	10%	4%	1%	-9%	-1%	-1%	**4%**
	wave5	43%	-9%	4%	15%	-26%	-7%	-5%	**5%**
	mgrid	45%	13%	36%	0%	13%	-24%	25%	**36%**
	Spec95fp avg.	57%	9%	10%	4%	0%	-15%	5%	**10%**
	overall avg.	51%	7%	4%	4%	-5%	-8%	6%	**7%**

Table 1: Table with results for programs from the Olden and the SPEC95FP benchmarks: mem. stall=percentage of time the processor stalls waiting for the memory; mem. stall reduction=the percentage of memory stall time reduction after optimization; source CH speedup=the speedup if only source cache specifiers are used; target CH speedup=speedup if only target cache specifiers are used; missrate reduction=reduction in miss rate for the three cache levels; overall speedup=speedup resulting from reuse distance-based cache hint selection.

	miss rate reduction	
program	**static hints**	**dynamic hints**
vpenta(miss-rate=29.4%)	6%	6%
mxm (miss-rate=1.3%)	0%	0%
liv18 (miss-rate=15.62%)	0%	0%
cholesky (miss-rate=5.75%)	-22%	1%
jacobi (miss-rate=21.02%)	32%	32%
gauss-jordan (miss-rate=11.9%)	25%	34%
tomcatv (miss-rate=9.67%)	6%	9%
average	6.7%	11.7%

Table 2: The cache miss rates for a 4-way set associative 16KB cache with 32 bytes per line. The first column indicates the program under consideration. Next to the program name, between brackets, the cache miss rate for the program without cache hints is indicated. The second column indicates the relative number of cache misses for the program with cache hints per instruction, compared to the program without hints. The third column shows the relative number of cache misses with cache hints per access.

tions, data layout optimizations or hardware or software prefetching schemes[22, 21, 15, 13]. Most of the hardware modifications, loop transformations and data layout transformations aim at improving the cache locality of the program. The prefetching schemes aim at hiding the latency of cache misses with parallel instructions. The proposed cache hint selection scheme aims to do both: the source cache hints are used to hide the latency of the cache misses, while the target cache hints improve the replacement policy of the data cache.

The source cache hints allow the compiler to try to hide the latency of cache misses with parallel instructions, while fetching the data from a lower cache level. Most related research focusses on generating prefetch instructions to do this. However, prefetching requires extra prefetch instructions to be inserted in the program. In the source cache hint approach, the latency is hidden without inserting prefetch instructions. Similar techniques are proposed in [9] and [16]. In [9] the cache behavior of numerical programs is examined using miss traffic analysis. The detected cache miss latencies are hidden by techniques such as loop unrolling and shifting. In comparison, our technique also applies to non-numerical programs and the latencies are compensated by scheduling low level instructions. In [16], load instructions are classified into normal, list and stride access. List and stride accesses are maximally hidden by the compiler because they cause most cache misses. However the classification of memory accesses in two groups is very coarse. The reuse distance provides a more accurate way to measure the data locality, and as such permits the compiler to generate a more balanced schedule.

Work related to target cache hints is found in [12], [20],[25] and [24].

In [12], keep and kill instructions are proposed. The keep instruction locks data into the cache, while the kill instruction indicates it as the first candidate to be replaced. Jain et al. also proof under which conditions the keep and kill instructions improve the cache hit rate. In [24], it is proposed to extend each cache line with an EM(Evict Me)-bit. The bit is set by software, based on a locality analysis. If the bit is set, that cache line is the first candidate to be

evicted from the cache. In [20], a cache with 3 modules is presented. The modules are optimized respectively for spatial, temporal and spatial-temporal locality. The compiler indicates in which module the data should be cached, based upon compiler analysis or a profiling step. These approaches all suggest interesting modifications to the cache hardware, which allow the compiler to improve the cache replacement policy. However, the proposed modifications are not available in present day architectures. The advantage of our approach is that it uses cache hints available in existing processors. The results show that the presented cache hint selection scheme is able to increase the performance on real hardware.

The analytical approach to calculate reuse distances is related to the field of analytical calculation of cache behavior. Most of the previous work on analytical cache behavior calculation is based on reuse vectors[8, 19, 23], which do not capture all reuses in loops. Therefore, these methods result in an inexact estimate of the cache behavior. Recently, Chatterjee et al.[6] proposed a technique to exactly calculate the cache behavior. While their technique leads to the exact calculation for low-associative caches, the proposed cache equations do not allow to efficiently obtain cache behavior for high associativity ($\geq$ 4 way set associative). In contrast, the calculation of reuse distances allows to calculate the cache behavior of fully associative caches. Furthermore the proposed cache equations can be extended to also handle arbitrary caches.

8. CONCLUSION

EPIC architectures provide cache hints which allow the compiler to have more control and a better view on the data cache behavior. The source cache hints inform the compiler to hide long latency loads with parallel instructions, while the target cache hints enable an adapted replacement policy for data with low locality. In this work, a framework is proposed to generate both source and target cache hints from the reuse distance metric. Since the reuse distance indicates cache behavior irrespective of the cache size and associativity, it can be used to make caching decisions for all levels of cache simultaneously. In this paper, this property is exploited to select appropriate cache hints for multiple levels of cache.

Two methods were proposed to determine the reuse distances in the program, one based on profiling which statically assigns a cache hint to a memory instruction and one based on analytical calculation which allows to dynamically select the most appropriate hint. The advantage of the profiling-based method is that it works for all programs. The analytical calculation of reuse distances is applicable to loop-oriented code and has the advantage that the reuse distance is calculated independent of program input and for every single memory access. Furthermore, the analytical calculation allows to generate dynamic cache hints, i.e. different cache hints can be generated for different executions of the same original memory instruction. The experimental evaluation of the profile-based source and target cache hint selection on a number of pointer-intensive and numerical programs shows an average speed up of 7% with a maximum of 36%. The pointer-intensive programs profit most from the target cache hints, while for the numerical programs the source and target cache hints are of equal importance. Furthermore, it is shown for a number of loop-oriented programs that dynamic target cache hint selection can reduce cache misses better than static hints. On average, the static hints reduce the number of cache misses by 7%, while the dynamic cache hints reduce the cache misses by 12%. Dynamic target cache hints are especially valuable in programs where the same instruction can exhibit wildly varying locality. Therefore, in the future, we will further investigate ways to efficiently select cache hints dynamically, also for non-loop oriented programs.

9. REFERENCES

[1] K. Beyls and E. D'Hollander. Compiler generated multithreading to alleviate memory latency. *Journal of Universal Computer Science, special issue on Multithreaded Processors and Chip-Multiprocessors*, 6(10):968–993, oct 2000.

[2] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, pages 617–662, Aug 2001.

[3] I. R. Bratt, A. Settle, and D. A. Connors. Predicate-based transformations to eliminate control and data-irrelevant cache misses. In *Proceedings of EPIC-1*, pages 15–22, 2001.

[4] M. Brehob and R. J. Enbody. An analytic model of locality and caching. Technical Report MSU-CSE-99-31, Department of Computer Science, Michigan State University, East Lansing, Michigan, August 1999.

[5] J. F. Cantin and M. D. Hill. Cache performance for selected SPEC CPU2000 benchmarks. *Computer Architecture News (CAN)*, September 2001.

[6] S. Chatterjee, E. Parker, P. Hanlon, and A.R.Lebeck. Exact analysis of the cache behavior of nested loops. In *PLDI*, pages 286–297, 2001.

[7] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*, pages 278–285. ACM, May 1996.

[8] S. Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behaviour*. PhD thesis, Princeton University, November 1999.

[9] P. Grun, N. Dutt, and A. Nicolau. MIST: An algorithm for memory miss traffic management. In *International Conference on Computer Aided Design*, pages 431–437, 2000.

[10] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.

[11] *IA-64 Application Developer's Architecture Guide*, May 1999.

[12] P. Jain, S. Devadas, D. Engels, and L. Rudolph. Software-assisted replacement mechanisms for embedded systems. In *International Conference on Computer Aided Design*, pages 119–126, nov 2001.

[13] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and E. Ayguade. An integer linear programming approach for optimizing cache locality. In *Proceedings of the 1999 Conference on Supercomputing*, pages 500–509, 1999.

[14] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL_PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard, February 2000.

[15] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[16] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *MICRO'95*, pages 243–248, Ann Arbor, Michigan, Nov. 29–Dec. 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[17] M. Presburger. Über die Volständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves*, pages 91–101, 1929. Warsaw, Poland.

[18] W. Pugh. Counting solutions to Presburger formulas: How and why. *ACM SIGPLAN Notices*, 29(6):121–134, jun 1994.

[19] J. Sánchez and A. González. Fast, accurate and flexible data locality analysis. In *PACT '98*, pages 124–129, Paris, France, Oct. 12–18, 1998. IEEE Computer Society Press.

[20] J. Sanchez and A. Gonzalez. A locality sensitive multi-module cache with explicit management. In *Proceedings of the 1999 Conference on Supercomputing*, ACM SIGARCH, pages 51–59, N.Y., June 20–25 1999. ACM Press.

[21] A. Seznec and F. Bodin. Skewed-associative caches. In *Proceedings of PARLE '93*, Lecture Notes in Computer Science, pages 305–316, Munich, Germany, June 14–17, 1993. Springer-Verlag.

[22] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[23] X. Vera and J. Xue. Let's study whole-program cache behavior analytically. In *High-Performance Computer Architecture (HPCA'02)*, pages 175–186, Feb 2002.

[24] Z. Wang, K. McKinley, A. Rosenberg, and C. Weems. Using the compiler to improve cache replacement decisions. In *PACT'02*, September 2002.

[25] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In *HPCA-6*, pages 49–60, Jan. 8–12, 2000.

[26] F. Zhang. *The FPT Parallel Programming Environment*. PhD thesis, Ghent University, 1996.

Performance Advantage of the Register Stack in Intel® Itanium™ Processors

Ryan Rakvic, Ed Grochowski, Bryan Black, Murali Annavaram, Trung Diep, and John P. Shen

Microprocessor Research, Intel Labs (MRL)
2200 Mission College Blvd
Santa Clara, CA 95052
ryan.n.rakvic@intel.com

Abstract

The Intel® Itanium™ architecture provides a virtual register stack of unlimited size for use by software. New virtual registers are allocated on a procedure call and deallocated on return. Itanium processors implement the register stack by means of a large physical register file, a mapping from virtual to physical registers, and a Register Stack Engine (RSE) that saves and restores the contents of the physical registers to memory without explicit program intervention. The combination of these features significantly reduces the number of loads and stores required to save registers across procedure calls compared to a conventional architecture. In this paper, we show that the Itanium register stack reduces load and store traffic to the stack by at least a factor of three across select SpecInt2000 and Oracle database benchmarks. Furthermore, we examine the effects of the register stack on data cache miss rates and program execution time. When compared to a conventional architecture, the Itanium architecture on average achieves 7%-8.3% and 10.2%-12% performance advantage on in-order and out-of-order processor models, respectively, as a result of the register stack. Finally we analyze the vitality of stack loads and show that in general few stack loads are vital in an in-order model. However, a larger percentage of stack loads become vital in the out-of-order model leading to a greater performance benefit from the register stack.

1. Introduction

The Intel® Itanium™ architecture contains a number of innovative features designed to speed up program execution. These features include:

1. Explicit parallelism: a mechanism whereby the compiler identifies groups of independent instructions for parallel execution by hardware
2. Control speculation: a mechanism to move loads and their dependents ahead of conditional branches
3. Data speculation: a mechanism to move loads and their dependents ahead of potentially conflicting stores
4. Predication: conditional execution of instructions, thereby avoiding mispredicted branch penalties
5. Register stack: a large physical register file organized as a stack with hardware to automatically save and restore registers to memory

An overview of the Itanium architecture may be found in [1] and [2]. A detailed description may be found in [3]. As the Itanium architecture is very new, we are just beginning to understand the performance implications of each of these features. An in-depth analysis of the effectiveness of predication was presented in [4] showing very limited performance advantage. In this paper, we focus on the register stack. We describe the features of the register stack, while highlighting both the positives and negatives. We show that the Itanium register stack provides an overall performance advantage when compared to a similar processor without one. This is not a paper proposing a new idea for a future design, but an evaluation of a known technique that has actually been implemented in a real machine for a new architecture. The goal of this paper is to assess how well the Itanium register stack performs and to explain why. To our knowledge, no previous technical paper has provided a detailed performance evaluation of a register stack architecture.

The outline of this paper is as follows. Section 2 presents an overview of the Itanium register stack. Section 3 presents the simulation methodology used in this paper. We then compare the Itanium register stack to a conventional register file, noting the register stack's effects on dynamic instruction counts in Section 4, cache miss rates in Section 5, and execution time in Section 6. Finally, we analyze the vitality of stack loads in Section 7.

2. Itanium Register Stack

Overlapping register windows have long been used in RISC architectures to store the local variables required by multiple procedures in a large on-chip register file [5,6,7]. Fixed size, two size, and variable size register windows have been studied in [8], and hardware register windows have been compared against software register allocation in [9]. Prior to the Itanium Processor Family (IPF), the only major commercial microprocessors that employ register windows were the SPARC microprocessors. This section presents an overview of the Itanium register stack.

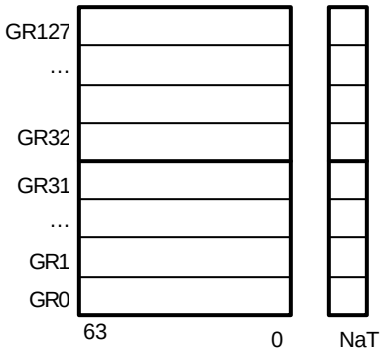


Figure 1 Itanium Integer Register File.

The Itanium architecture specifies 128 architected 64-bit integer registers as shown in Figure 1. Each register stores 64-bits of integer data along with a single NaT (*Not a Thing*) bit that indicates a deferred exception has occurred during control speculation. The 128 architected integer registers are divided into two groups: 32 static registers GR[0:31] and 96 stacked registers GR[32:127]. The static registers are available to all procedures while the stacked registers are allocated on demand to each procedure. GR0 is always read as zero. A subset of the stacked registers may be used as rotating registers during software pipelined loops; the rotating subset is considered part of a procedure's local variables.

Register allocation works as follows. Each procedure is responsible for allocating a stack frame consisting of input parameters, local variables, and output parameters. This is accomplished via the *alloc* instruction. Stack frames overlap so that the registers holding the output parameters of the caller become the input parameters of the callee. The overlap is illustrated in Figure 2.

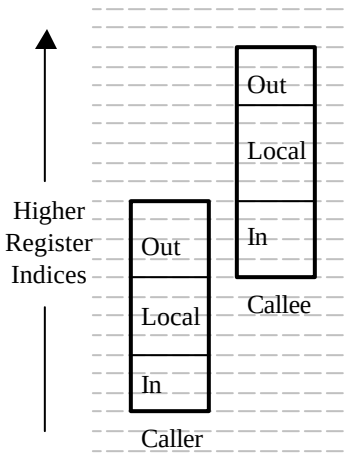


Figure 2 Overlapping Stack Frames.

Due to the overlap, each procedure needs to specify on an *alloc*:

1. The number of registers needed for input parameters and local variables
2. The total number of registers needed by the procedure's stack frame, equal to the sum of input, local, and output variables

Each procedure's stack frame may contain any number of registers up to a maximum of 96. A new stack frame is created by a procedure call and the associated *alloc*. The frame is automatically deallocated on the procedure return. Unlike stacks in other computer architectures, the register stack in the Itanium architecture grows *upwards* towards higher register indices and higher memory locations.

While the Itanium architecture specifies 96 architected stacked registers, the actual number of physical stacked registers in an implementation may be larger than 96. The physical register stack must contain at least 96 registers in order to support the ability of a procedure to allocate up to a maximum of 96 registers in its stack frame. The physical register stack in the current Itanium processors contains the minimum 96 stacked registers. Future implementations may have a larger physical register stack.

What happens when the total number of registers allocated in all stack frames exceeds the size of the physical register stack? The Itanium architecture defines a hardware Register Stack Engine (RSE), which transfers the contents of the physical registers to and from memory in order to create the illusion of an infinite-size register stack. The area of memory reserved for this purpose is called the *backing store*; the act of storing registers to the backing store is called a *spill*, and the act of loading registers from the backing store is called a *fill*. The relationship between the register stack and the backing store is illustrated in Figure 3.

The physical register stack is partitioned into three regions. The invalid region contains not-yet-allocated registers. The active region is used by the current procedure to hold input parameters, local variables, and output parameters. Below the active region is the dirty region. These registers hold the stack frames belonging to the current procedure's *callers* that have not been written to memory. Once spilled to memory, a register becomes part of the invalid region for subsequent use by the current procedure's *callees*.

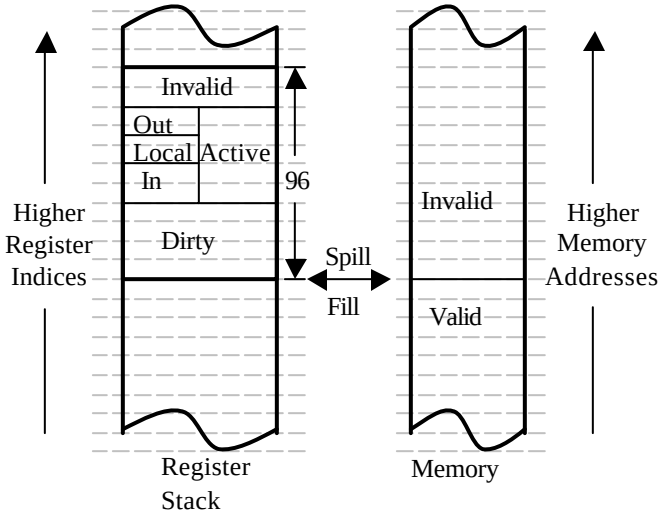


Figure 3 Relationship between Register Stack and the Backing Store in memory.

Even though the physical register stack has a fixed size, the RSE's job is to maintain the illusion of an infinite-size register stack by spilling and filling registers to memory without explicit control by the application program. The physical register stack is maintained as a circular buffer by the RSE and functions as a "window" into the infinite-sized virtual register stack. One RSE algorithm is as follows:

1. When a procedure allocates a new stack frame, if the top of the frame (active region) extends above the top of the physical register stack window, then the window is moved up by spilling some dirty registers to the backing store. These dirty registers belong to the current procedure's *callers*.
2. After a procedure returns and its stack frame is discarded, if the bottom of the caller's frame (now the active region) extends below the bottom of the physical register stack window, then the window is moved down by filling registers from the backing store. These registers belong to the current procedure.
3. Procedure calls, *allocs*, and returns within the physical register stack window do not need spills/fills.

Note that RSE spills and fills to the backing store are subject to address translation and may incur memory exceptions. The Itanium architecture requires that exceptions resulting from RSE activities be precisely reported.

Figure 4 illustrates the actions of the RSE during the execution of a 20K instruction excerpt from *gcc*. As the program's call depth changes, the RSE moves the location of the register window so that the current frame is always mapped within the window (containing 96 physical registers in this simulation). Moving the window up generates spills while moving the window down generates fills. Not all changes in call depth require movement of the register window. The ability to keep the register window unchanged across multiple procedure calls leads to a reduction in load/store traffic. The rest of this paper analyzes this effect, and shows the overall impact of the register stack.

3. Simulation Methodology

This section describes the simulation methodology used in this paper. For workloads, we select from the SPEC CPU2000 integer suite seven benchmarks that exhibit reasonable amounts of procedure call activity: *crafty*, *gap*, *gcc*, *gzip*, *parser*, *perlbmk*, and *vortex*. In addition, we include the *Oracle Database Benchmark* [10] as an example of a large transaction processing workload.

All benchmarks are compiled with the Intel Electron compiler for the Itanium Processor Family. We run 250M instruction samples of each benchmark using two Itanium instruction set simulators: SoftSDV [22] for the CPU2000 benchmarks and Simics [23] for the *Oracle* benchmark. The CPU2000 benchmarks primarily consist of applications code, whereas *Oracle* has significant contributions from both the application and operating system that are included in our trace. The results of the instruction set simulators are fed to a cycle-accurate microarchitecture performance simulator that implements the machine configuration summarized in Table 1.

Pipeline Depth	8 cycle branch misprediction penalty
Fetch Width	2 bundles
Branch Predictor	512 entry BTB 4096 entry PHT 8 bit local history
Register Files	128 Integer Registers 128 FP Registers 64 Predicate Registers
Execution Resources	6 integer ALUs 2 load 2 store
Cache Structure	L1I: 16K 4-way, 1 cycle latency L1D: 16K 4-way, 1 cycle latency L2: 256K 4-way, 6 cycle latency L3: 3072K 12-way, 12 cycle latency All caches have 64 byte lines
Main Memory	100 cycle latency

Table 1 Research Itanium Machine Model Parameters.

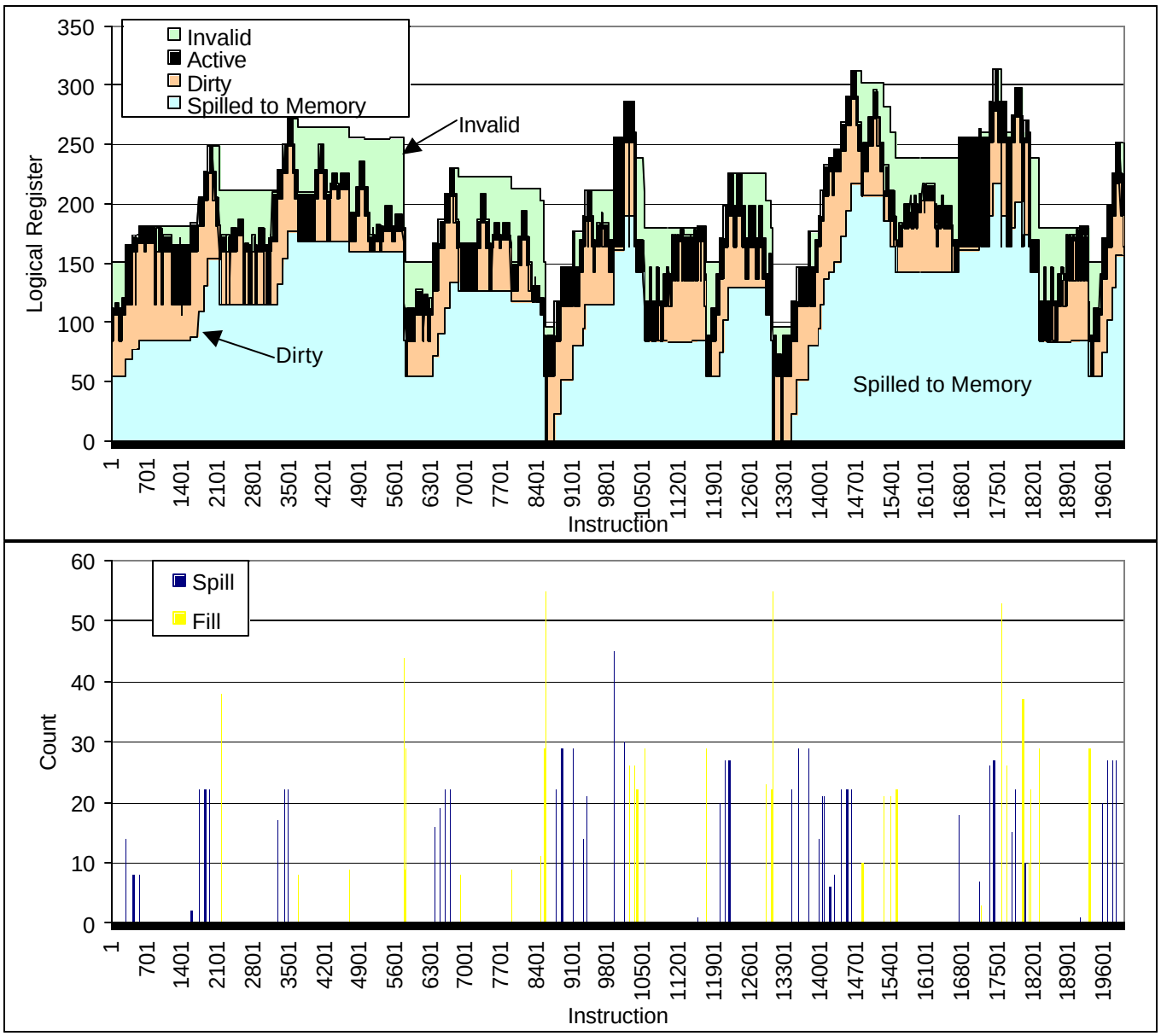


Figure 4 Snapshot of the Itanium Register Stack in Operation.

While the activities of the register stack may be measured using the performance-monitoring features of the Itanium processor, we use simulation to enable studies of arbitrary register stack configurations. We consider both in-order and out-of-order implementations of the Itanium architecture. Although current IPF implementations are in-order, out-of-order IPF processors may be implemented in the future using the techniques described in [11].

To compare the performance impact of the register stack, we simulate two distinct machine models:

1. One with the register stack (IPF)
2. One without the register stack (conventional).

For the conventional model, we modified our simulator's RSE to perform saves and restores of all registers used for local variables in each procedure. The resulting memory traffic is similar to what a conventional architecture with 128 architected registers and callee-save conventions would experience. In the next section we begin by showing the increase in memory traffic in terms of extra loads/stores for the conventional machine relative to the IPF machine with the register stack.

	crafty	gap	gcc	gzip	parser	perlbmk	vortex	oracle	total
Loads	13.13%	16.07%	14.72%	15.65%	17.78%	19.96%	15.90%	17.00%	16.21%
Stores	2.98%	6.93%	5.24%	8.09%	3.59%	7.82%	8.28%	8.12%	6.31%
Calls	0.27%	0.64%	1.14%	0.28%	0.68%	1.72%	1.12%	0.84%	0.83%
IPF-96 spills+fills	10.42%	0.30%	6.29%	0.00%	1.29%	0.29%	2.75%	7.74%	3.92%
IPF-128 spills+fills	5.11%	0.06%	1.52%	0.00%	0.67%	0.00%	1.22%	4.70%	1.80%
IPF-192 spills+fills	1.82%	0.02%	0.25%	0.00%	0.18%	0.00%	0.33%	2.13%	0.64%
Conventional saves+restores	16.49%	18.38%	22.01%	8.49%	11.62%	21.46%	8.30%	14.63%	15.26%

Table 2 IPF and Conventional Dynamic Instruction Counts.

4. Dynamic Instruction Counts

Table 2 presents dynamic instruction counts for the CPU2000 and *Oracle* benchmarks. The instruction counts are expressed as a percentage of the total non-NOP retired instructions for the *IPF-96* configuration. On IPF processors, NOPs account for 28% of all retired instructions. The NOPs are the result of the instruction templates, which allow only certain instruction types in each of the three instruction slots, and branch targets that are aligned to boundaries of three-instruction bundles. Total non-NOP retired instructions for the IPF-96 configuration are used as the denominator for all configurations. The number of RSE spills is equal to the number of RSE fills; the results in the table show the total of the two. Similarly, the number of register saves is equal to the number of register restores, and is half of the total shown. We refer to IPF stack stores and loads as spills and fills respectively, and to conventional architecture stack stores and loads as saves and restores respectively.

As shown in Table 2, loads account for 16.2% of all retired instructions; stores account for 6.3%, and procedure calls account for 0.8%. The percentage of loads and stores (22%) in the Itanium architecture is significantly lower than other architectures due to the large physical register file and the register stack. The number of RSE spills and fills with 96 physical stacked registers is shown in the *IPF-96* row. RSE spills account for only 2.0% of all instructions, and RSE fills account for another 2.0%. Although the contribution of the RSE to the total instruction count is small, there is considerable variation among the benchmarks. *Crafty, gcc,* and *Oracle* contain high levels of RSE activity, while *gzip* contains essentially no RSE activity.

Overall, having 96 physical stacked registers is quite effective in minimizing spill/fill traffic. As shown in the *IPF-128* row, increasing the number of physical stacked registers from 96 to 128 can reduce the number of spills and fills by a factor of two to just 1.8%. Further increasing the physical register stack size to 192 can reduce the spill/fill traffic to 0.6%. This indicates the true effectiveness of the register stack.

As stated in Section 3, we also simulate the number of stack register saves and restores that would be required in a conventional architecture. This is accomplished by modifying our simulator's RSE to save and restore all local registers in each procedure's stack frame as indicated by the *alloc* instruction. We assume that only local variables need be saved and restored. Registers containing input and output parameters, as well as the 32 static registers, are not preserved by the RSE. The resulting memory traffic approximates the behavior of a conventional architecture with 128 architected registers and callee-save conventions.

We observe that in a conventional architecture, saves and restores account for 15.3% of all instructions. Comparing the *IPF-96* against the *Conventional* configuration, we see that the Itanium register stack is successful in reducing the memory traffic required to save and restore registers across procedure calls by a factor of 3.9. With *IPF-128* this factor becomes 8.5. Comparing the total number of loads and stores performed by the *IPF-96* and *Conventional* machines, we note that *IPF-96* would have experienced a 31% increase in load traffic and 68% increase in store traffic in the absence of the register stack. In subsequent sections, we'll evaluate the Itanium architecture using the *IPF-96* and *Conventional* configurations, referring to these configurations simply as *IPF* and *Conventional* respectively.

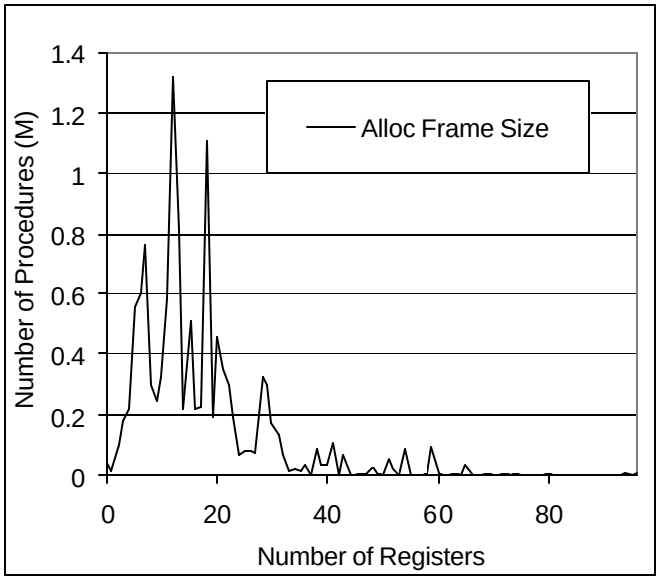


Figure 5 Number of Dynamic Procedures vs Number of Stacked Registers.

Unlike the fixed-size register windows in the SPARC architecture, the Itanium register stack allows the flexible allocation of arbitrary number of registers for each stack frame. Figure 5 presents the distribution of the total number of registers allocated in each procedure's stack frame. This data is summed across the CPU2000 and *Oracle* benchmarks. The most frequently occurring stack frame allocates 12 registers (in addition to the 32 static registers not included in the graph). Furthermore, 90% of all procedures allocate fewer than 30 registers. This data indicates that the 96 physical stacked registers in the present IPF implementations can be expected to hold the stack frames for multiple procedures and keep the number of spills and fills to a minimum.

One limitation of the present compiler is that it allocates once (at the beginning of each procedure) the maximum number of registers required by all possible control flow paths through that procedure. Depending on the actual control flow path, not all allocated registers may be used by the procedure. Table 3 presents the ratio of the registers used by the actual control flow versus the number allocated at the beginning. On average, 76% of all allocated registers are used by the actual control flow. In both the Itanium and conventional architectures, it is possible for the compiler to implement either one allocation (or block of saves and restores) per procedure, or multiple finer-granularity allocations (or blocks of saves and restores) at different points within the procedure. For our comparison, we assume one allocation per procedure for both the *IPF* and *Conventional* configurations.

crafty	gap	gcc	gzip	parser	vortex	average
78.7%	59.8%	69.1%	80.1%	90.2%	84.3%	75.8%

Table 3 Ratio of Demand to Allocated Register Usage.

As previously stated, the Itanium architecture specifies a minimum of 96 physical stacked registers. Figure 6 shows the effects of varying the number of physical stacked registers on the number of RSE spills. As shown in Figure 6, the number of spills (and corresponding number of fills) can be further reduced, by increasing the number of physical stacked registers beyond 96. The number of spills increases dramatically with fewer than 96 physical registers.

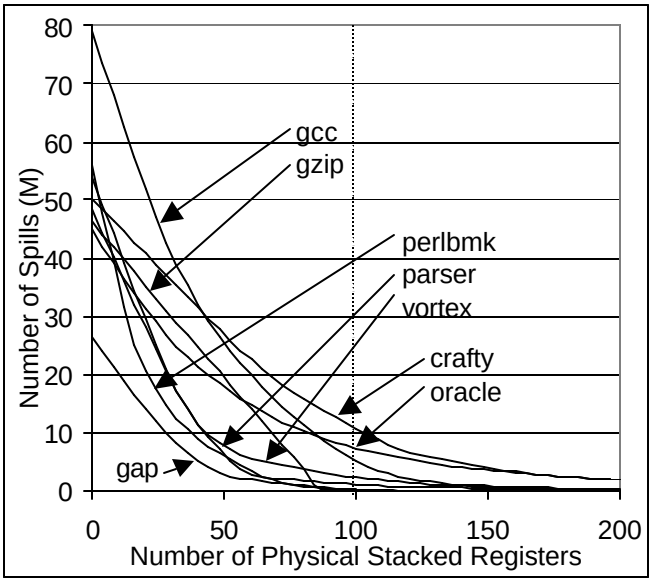


Figure 6 Varying the Number of Physical Stacked Registers.

5. Cache Impact of the Register Stack

This section examines the effects of the Itanium register stack on data cache miss rates and the number of misses. It is well known that the performance of a microprocessor is highly dependent on the performance of its cache memory. Most try to design a Data Level 1 Cache (L1D) that maintains a low miss rate with very low latency [12]. Current approaches include using streaming buffers,

victim caches [13], alternative cache indexing schemes [14], etc. [15].

Section 4 shows that the register stack eliminates many stack loads and stores. We first analyze cache impact of the register stack by detailing the miss rate of the loads that are accessing the stack (we refer to IPF stack loads as *fills* and conventional architecture stack loads as *restores*). These stack loads share the L1D cache with non-stack (heap) loads, but for now we separate their miss rates. Figure 7 presents the L1D miss rates of only these stack loads. (Recall that the L1D cache is fixed at 16KB.) As can be seen, the miss rate for the stack loads is very low for both configurations. On average, the miss rate for stack loads is below 3%. For most of the benchmarks, the conventional configuration has a lower miss rate than the IPF configuration. The conventional configuration has many more stack references, and this leads to better locality and a lower miss rate.

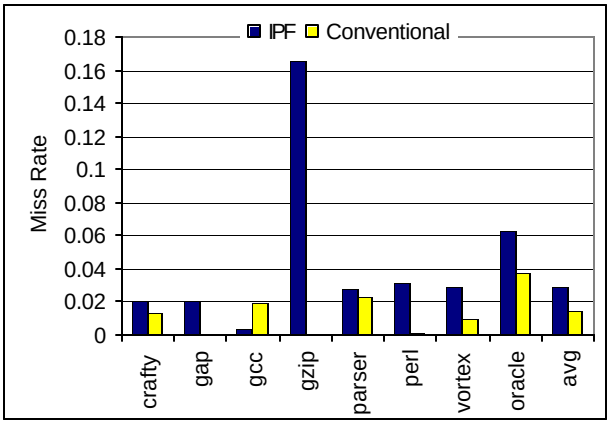


Figure 7 L1D Cache Miss Rate of Stack Loads.

Although the miss rate is lower for the conventional configuration, the total number of misses from stack loads is greater for most of the benchmarks. Figure 8 illustrates the overall number of misses generated by stack loads normalized to the IPF configuration. For example, for *perl*, the conventional configuration has over 4.3 times the number of stack load misses than the IPF configuration. On average, the conventional configuration has approximately 90% more overall stack load misses than the IPF configuration. However, as we saw in Figure 7, the stack loads have a very low miss rate and hence low overall misses. Therefore, the register stack's cache impact does not significantly impact overall performance.

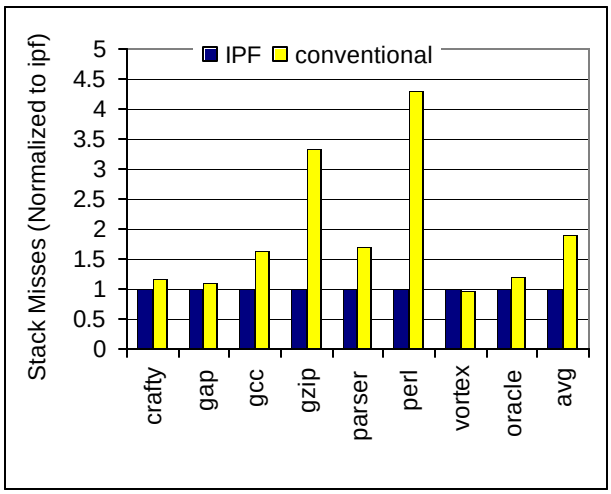


Figure 8 Overall Stack Load Misses.

We now analyze the impact the register stack has on the overall number of load misses (stack & heap). Figure 9 presents the overall number of load (stack & heap) misses (normalized to IPF) for all the benchmarks. The conventional configuration introduces more stack loads accessing more data space, and therefore increases the total number of misses. Although the stack loads' miss rates are low, they still add more overall misses since a majority of those loads do not exist in the IPF configuration. Also, the extra stack loads add extra data contention to the L1D. For example, for the benchmarks *crafty* and *Oracle*, they experience a similar number of stack misses (Figure 8), but the total number of L1D misses increases significantly. However, it can be seen that the total number of misses does increase, but not significantly for most of the benchmarks because the

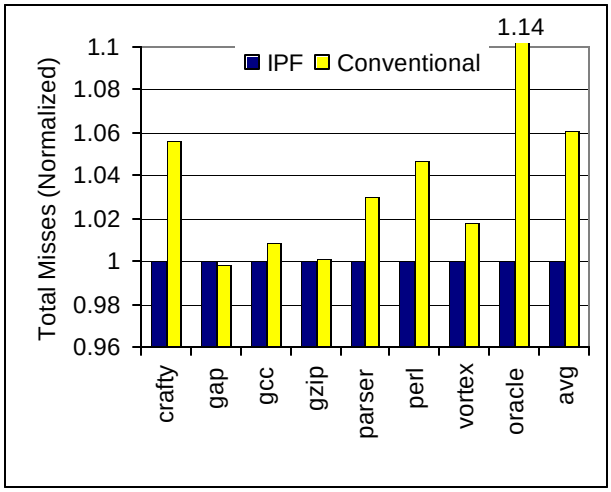


Figure 9 L1D Total Load Misses.

working set of the stack is miniscule relative to the 16KB L1D. Furthermore, the total number of misses of the L1D is small to begin with, and the overall cache impact of the register stack is also small.

6. Performance Advantage of the Register Stack

Up to this point, we have considered only secondary effects of the register stack in IPF processors. Now we consider overall performance impact of the register stack in terms of speedup. Once again we compare the Itanium-96 (IPF) model with a similar machine with no register stack (conventional).

Figure 10 presents the speedup, in terms of total execution cycles, of the IPF configuration compared to the conventional configuration. Both configurations are simulated on an in-order model, and assumed to have the parameters of Table 1. The performance benefit of the register stack ranges from 1.7% - 11.9% averaging 7%.

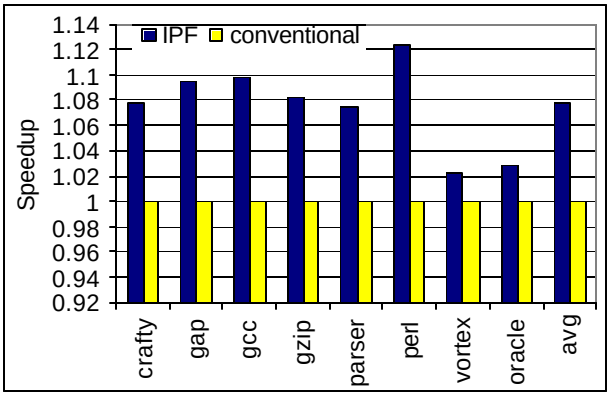


Figure 10 IPF-96 Performance Advantage (In-Order).

The register stack in the Itanium architecture reduces the total number of stack references, and hence provides a performance advantage compared to the same machine without a register stack. The performance advantage of the register stack can be attributed to the reduction in overall instructions. Table 2 presents the overall difference in instruction counts. In these in-order machines, there is strong correlation between instruction count and overall execution time since the in-order machine does not overlap the execution of stack loads and stores with a program's other instructions. However for the *Oracle* benchmark, the performance impact does not directly correlate with the increase in spill/fill count. *Oracle* spends a lot of time waiting for memory (and exhibits a lower IPC), and the increase in overall instructions has less of an effect on performance.

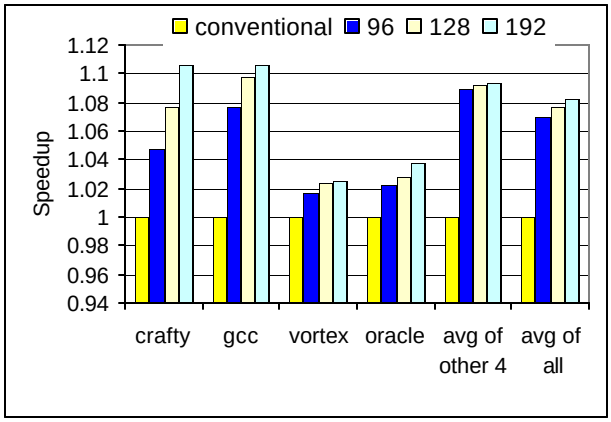


Figure 11 Register Stack Size Impact on Performance (In-Order).

The performance numbers presented thus far assume a physical register stack size of 96. This number can be increased in actual implementations. As Table 2 illustrates, a few of the benchmarks could potentially benefit from an increase in physical register stack size. Figure 11 provides the speedup achieved for various (96, 128, 192) stack sizes and compares them to the conventional machine. For the benchmarks *crafty*, *gcc*, *vortex*, and *Oracle*, an increase in the stack size from 96 to 128 increases performance greatly. Furthermore, for *crafty* and *Oracle*, an increase in the stack size to 192 still significantly increases performance. For instance, *Oracle*'s performance gain over the conventional machine increases from 2.2% (96 stacked registers) to 3.8% (192 stacked registers). We can conclude that *Oracle* is using a lot of registers in a short amount of time. For the other 4 benchmarks (*gap*, *gzip*, *parser*, *perl*), a register stack size of 96 is more than enough to reduce spills/fills and hence performance is not increased significantly with an increase in register stack size. The overall average performance benefit of the register stack ranges from 7% (96 stacked registers) to 8.3% (192 stacked registers) for the in-order machine.

The register stack is an architectural feature that can be applied to an in-order or an out-of-order implementation. So far we have presented numbers for an in-order model. Now we present the performance advantage of the register stack in the context of an out-of-order model. Once again, we show speedup for the two configurations (IPF and conventional). Figure 12 presents the speedup for the benchmarks. The performance advantage of the register stack is slightly higher in the out-of-order model compared to the in-order model. In the out-of-order model, the register stack averages a 10.2% increase in performance for all benchmarks for *IPF-96*. For *IPF-192* (not in figure), the performance gain is 12%.

One would expect the out-of-order model to mask some of the performance advantage of the register stack because independent instructions can execute and use the available resources while the stack loads and stores execute concurrently. However, as shown in Figure 12, the out-of-order model does not reduce the performance advantage of the register stack. In fact, the opposite is true: the register stack provides a greater performance benefit in the out-of-order machine. As we will see in Section 7, the extra stack loads in the conventional model are vital instructions [17], and dependent instructions are forced to wait for these stack loads to execute.

We see in Section 5 that the cache effects of the register stack are minimal. The extra stack loads do not increase the overall number of misses experienced by the L1D significantly. Additionally, we find that the latency of the L1D has minimal impact on the extra stack loads. However, the bandwidth for executing stack loads and stores does have an impact on the performance advantage of the register stack.

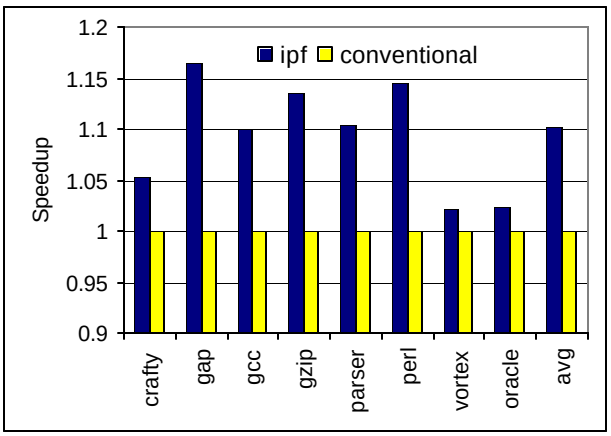


Figure 12 IPF-96 Performance Advantage (Out-Of-Order).

[16] cleverly provided high-bandwidth and low-latency caches by decoupling accesses of stack variables from heap variables. The decoupled caches provide more bandwidth when stack variable accesses are intermingled with non-stack accesses. The design requires identification of local accesses either statically or dynamically. Our two configurations use the L1D for both stack and heap data. So far we fixed the number of ports for the L1D at four (2 Read & 2 Write). This allows for 2 loads and 2 stores to be executed in parallel. Now we simulate the same two configurations (IPF and conventional) with the number of ports for the L1D set at two (1 Read & 1 Write). This allows only 1 load and 1 store to be executed in parallel. Figure 13 presents the speedup of the register stack under these constraints. Comparing Figure 10 and Figure 13 it can be seen that the performance advantage of the register stack increases when the bandwidth of the L1D is reduced. For the in-order machine, the overall benefit of the register stack increases from 7% to 9% on average. When peak bandwidth is limited, the conventional configuration suffers a bigger performance penalty due to the greater number of stack loads.

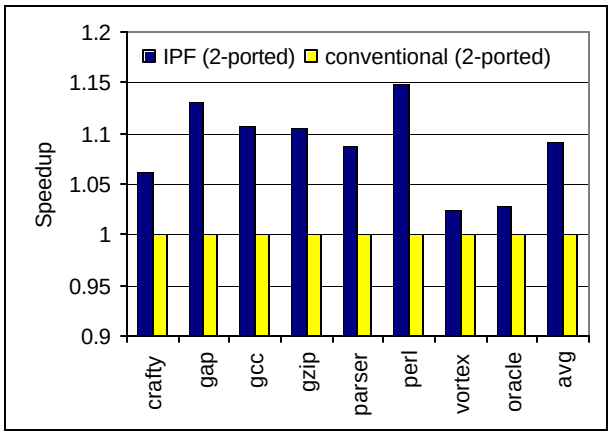


Figure 13 Load/Store Bandwidth Analysis (In-Order).

We also simulated the 2-ported configurations with an out-of-order model. Similar to the in-order model, the performance advantage of the register stack increases when the bandwidth of the L1D is limited. The performance advantage grows from 10.2% (4-ported) to 12.4% (2-ported) on average.

The performance advantage of the register stack is seen in both in-order and out-of-order machine models. The major contributor to the overall speedup is the reduced instruction count. The size of the register stack does not significantly impact the miss behavior of the L1D. However, the bandwidth of the L1D does have impact on non-register stack configurations. The out-of-order model consistently has bigger impact from the register stack. The following section explains this result.

7. Vitality of Stack Loads

[18][19][20] introduced the notion of load importance or criticality. [17] defined the concept of load *vitality*. Loads were classified into two categories: vital and non-vital. Vital loads are loads that must be executed as quickly as possible in order to maximize performance. Non-vital loads are loads that are less vital to program execution and for various reasons do not require immediate execution in order to maximize overall performance.

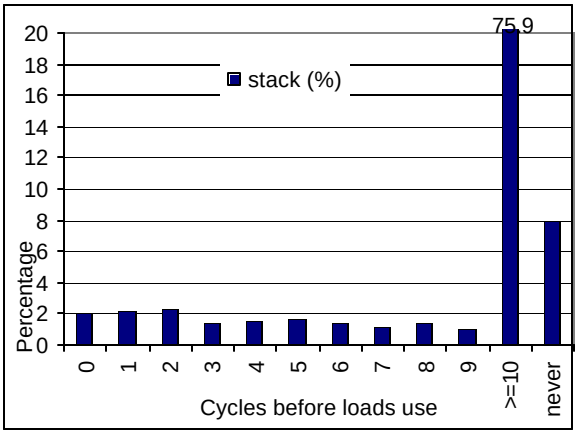


Figure 14 Dependence Distance of Stack Loads (In-Order).

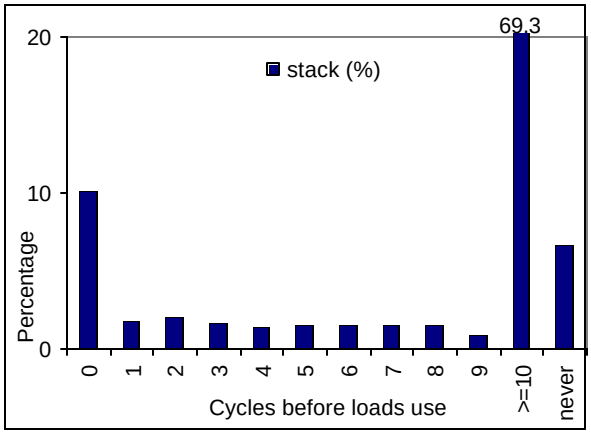


Figure 15 Dependence Distance of Stack Loads (Out-Of-Order Machine).

We measure vitality of the stack loads by counting the number of cycles between the execution of the stack load and the execution of the earliest dependent instruction. Figure 14 shows vitality for stack loads for the in-order conventional configuration. The x-axis represents the number of cycles between the load finishing execution and the earliest dependent instruction using its data. The "*0*" bar (first bar) represents the % of vital instructions. For example, 2.0% of the stack loads' data are used immediately and hence these are considered vital. The *>=10* bar represents data which are used in 10 cycles or greater, and the data which are never used are collected in the *never* category. We present the vitality for the conventional configuration in order to analyze the stack loads that the register stack successfully removes. As we can see, the stack loads are mostly non-vital (98% are non-vital). Therefore, the cache latency of stack loads is not a limiter in terms of performance. The loads that are eliminated by the register stack are mostly non-vital in the context of an in-order model.

Figure 15 shows the vitality of stack loads for an out-of-order model. Unlike the in-order model, 10% of stack loads are now vital. Section 6 shows that the performance advantage of the register stack is greater for an out-of-order model. More stack loads are now vital to performance because the out-of-order machine has moved up the dependent instructions. The register stack eliminates these vital stack loads in the out-of-order model and hence the IPF configuration outperforms the conventional configuration by a larger degree.

8. Conclusion

The goal of this work is to assess how well this technique actually performs for IPF machines and to explain why. In this paper, we examine the effects of the Itanium register stack on dynamic instruction counts, data cache miss rates, program execution time, and load instruction vitality. We conclude that the Itanium register stack with 96 physical stacked registers is successful in reducing the number of stack loads and stores by at least a factor of three compared to a conventional architecture. Furthermore, in the absence of the register stack, the Itanium architecture would have experienced a 31% increase in overall load traffic and 68% increase in overall store traffic. We show the register stack provides a 7%-8.3% and 10.2%-12% performance advantage in in-order and out-of-order machines, respectively. The higher performance advantage on an out-of-order machine is due to a greater number of stack loads becoming vital. Important benchmarks, such as *gcc* and *Oracle* have a large register window requirement. By making the physical register stack size larger than the current 96, almost all the spills and fills can be eliminated, resulting in even greater performance advantage. We believe that the register stack of the Itanium architecture is a very good feature, which will become more and more important as future workloads become more object oriented. Our preliminary measurements of Java workloads indicate that such workloads stress the register stack much more than the benchmarks in this study. Our future study will analyze such workloads.

9. References

[1] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, R. Zahir, “Introducing the IA-64 Architecture”, *IEEE Micro*, Sept-Oct 2000.

[2] R. Zahir, J. Ross, D. Morris, D. Hess, "OS and Compiler Considerations In the Design of the IA-64 Architecture", *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[3] Intel® Itanium™ Architecture Software Developer's Manual. *http://developer.intel.com/design/itanium/manuals/index.htm.*

[4] Y. Choi, A. Knies, L. Gerke, T. Ngai, “The Impact of If-Conversion and Branch Prediction on Program Execution on the Intel Itanium Processor”, *Proceedings of the 34rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2001.

[5] D. Patterson, C. Sequin, "RISC I", *Conference Proceedings of the Eighth Annual Symposium on Computer Architecture*, May 1981.

[6] T. Stanley, R. Wedig, "A Performance Analysis of Automatically Managed Top of Stack Buffers*", the 14th Annual International Symposium on Computer Architecture*, June 1987.

[7] R. Cmelik, S. Kong, D. Ditzel, E. Kelly, "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks", *ACM SIGARCH Computer Architecture News, Proceedings of the Fourth InternationalConference on Architectural Support for Programming Languages and Operating Systems*, April 1991, Volume 19, Issue 2.

[8] B. Furht, "A RISC Architecture with Two-Size, Overlapping Register Windows", *IEEE Micro*, Volume: 8 Issue: 2, April 1988, Pages 67-80.

[9] D. Wall, "Register Windows Vs. Register Allocation", *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988.

[10] M. Annavaram, T. Diep and J. Shen, “Branch Behavior of a Commercial OLTP Workload on Intel IA32 Processors”, *To appear in Proceedings of the International Conference on Computer Design*, September 2002.

[11] P. Wang, H. Wang, R. Kling, K. Ramakrishnan, J. Shen, “Register Renaming and Scheduling for Dynamic Execution of Predicated Code”, *In 7tt HPCA*, Jan 2001.

[12] M. D. Hill, “A Case for Direct-Mapped Caches”, *IEEE Computer*, pp. 25 - 40, Dec. 1988.

[13] R. Kessler, R. Jooss, A. Lebeck, and M. Hill, “Inexpensive implementations of set-associativity”, *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 17(3):131--139, 1989.

[14] A. Seznec, “A case for two-way skewed-associative caches”, *In 20th Annual International Symposium on computer Architecture*.

[15] L. John and A. Subramania, “Design and performance evaluation of a cache assist to implement selective caching”, *In International Conference on Computer Design.*

[16] S. Cho, P. Yew, and G. Lee, “Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor”, *Proc. of the 26th Int’l Symp. on Computer Architecture*.

[17] R. Rakvic, B. Black, D. Limaye, and John P. Shen, “Non-vital Loads”, *Proc.of the 8th International Conference on High-Performance Computer Architecture*, Feb. 2002.

[18] S. Srinivasan and A. Lebeck, “Load latency tolerance in dynamically scheduled processors.”, *in Proceedings of the Thirty-First International Symposium on Microarchitecture*, pp. 148--159, 1998.

[19] E. Tune, D. Liang, D. Tullsen, B. Calder, “Dynamic Prediction of Critical Path Instructions”, *In the Proceedings of the 7th High Performance of Computer Architecture*.

[20] B. A. Fields, S. Rubin and R. Bodik, “Focusing Processor Policies via Critical-Path Prediction”, *In proceedings of 28th International Symposium on Computer Architecture*.

[21] R.D.Weldon, S. Chang, H.Wang, G. Hoflehner, P. Wang, and J. P. Shen, “Quantitative Evaluation of the Register Stack Engine and Optimizations for Future Itanium Processors”, *In Interact-6 held in conjuction with the 8th International Conference on High-Performance Computer Architecture*, Feb. 2002.

[22] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, H. Wang, "SoftSDV: A Pre-silicon Software Development Environment for the IA-64 Architecture", *Intel Technology Journal*, Q4 1999, http://www.intel.com/technology/itj/q41999/articles/art_2.htm.

[23] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, "Simics: A Full System Simulation Platform", *Computer*, Volume 35, Issue 2, Feb. 2002, Pages 50-58.

Reducing the Physical Cost of Large Register Files in EPIC Architectures with Stacked Register Aliasing

Ron Arnold
Hewlett Packard Co.
rarnold@hp.com

Rohit Bhatia
Hewlett Packard Co.
rohit.bhatia@hp.com

Don Soltis
Hewlett Packard Co.
don.soltis@hp.com

Abstract

Large register files are a key feature of EPIC architectures. Such large register files coupled with wide instruction issue, also typical of EPIC machines, present an interesting design challenge. This paper presents a register aliasing technique to reduce the physical cost in terms of area, power and cycle time at the expense of a small performance loss. The register aliasing technique described can be applied to the hazard detection logic of a processor to reduce the complexity of this logic. Performance simulation results presented show a negligible loss for SpecINT95 suite and moderate loss for SpecFP95 suite. It is also shown how the physical register file size can actually be increased through the use of stacked register aliasing.

1. Introduction

EPIC architectures, such as the Itanium® architecture, provide key features to enhance instruction level parallelism and support explicit parallelism. Explicit parallelism is supported by large parallel execution resources and large register files. Speculation and predication support allow compilers to enhance ILP. Predication allows conditional execution without branches implying larger basic blocks. Both of these ILP enhancement features tend to require a larger set of registers. The Itanium® architecture provides a 128-entry integer and a 128-entry floating-point register file. Most conventional RISC architectures feature only 32-entry register files.

Although large register files are key ingredients of EPIC performance delivery, they present an interesting challenge for EPIC hardware designers in the face of ever shrinking processor cycle time. The design challenge is presented both in terms of register file access time and register hazard detection for pipelined processors. In addition, there is a need to support wide instruction issue.

The large number of physical registers in Itanium® architecture compounds an already difficult problem in the physical implementation, namely hazard detection. With increasing numbers of execution pipelines and increasing pipeline lengths, the number of in-flight data dependencies can be daunting. The complexity of detecting these myriad data hazards has begun to require non-traditional hardware methods.

In the register read (REG) pipeline stage all read-after-write (RAW) and write-after-write (WAW) data hazards must be detected in order resolve data dependency hazards. The processor must stall instructions whose source data will not be available as they enter the execution pipeline stage, as well as instructions whose results might otherwise be overwritten by older, longer latency, instructions. This is performed by comparing source and destination register identifiers (regids) in REG verses the destination regids of all data producers in later pipeline stages, up to the stage where the register file is actually written. These compares are not required for all producer-producer and producer-consumer combinations; the number is greatly reduced by data bypassing.

In the Itanium® 2 processor, while considerable bypassing is provided between stages and pipelines, there are still 720 regid compares required to detect the integer, or general register (GR) and floating-point register (FR) data hazards which can stall the instructions in the REG stage [3]. Each compare has up to 6 qualifiers, such as predicate values, instruction valids, and instruction types, which must be evaluated in order to determine if a register ID match constitutes a hazard. The magnitude of the task precludes single-cycle hazard detection using explicit numerical comparators and gating logic.

The producers and consumers in the REG stage must also be evaluated verses long-latency loads which have committed or retired but whose data is still unavailable. This is often performed using a load scoreboard, effectively an array of 1-bit registers with multiple read and write ports. There is one entry for each register in a given register file, which, when asserted, indicates a load

is pending a write to its corresponding register. Each consumer and producer must be provided a read port to determine pending writes to their targets. Each memory load pipeline must have a write port to set scoreboard entries as a load retires, and also a write port to clear scoreboard entries as data returned from the memory subsystem is finally written to the register file. The Itanium® 2 microarchitecture would require a 127x1 register file with 18 read ports and 6 write ports for the GRs alone. A similar scoreboard structure would be required for FR scoreboarding.

The Itanium® architecture provides 128 integer registers. Registers r0-r31 are always accessible and are known as static registers. The static registers are similar to the 32 general registers in most RISC architectures such as PA-RISC and PowerPC. The remaining 96 registers, r32-r127, are called stacked registers. The Itanium® architecture allows for more than 96 physical registers to be used to implement the stacked registers; however, only a maximum of 96 are visible architecturally at any given time [6].

This paper presents an aliasing technique, which is applied to the stacked registers only in an attempt to simplify the pipeline hazard detection logic with minimal performance cost. The physical complexity of the data hazard detection logic or the scoreboard is governed by the number of pipeline stages and the number of registers that need to be tracked. By aliasing the stacked register set, one can reduce the complexity and size of the hazard detection logic. This will lead to occasional false hazard detection and consequently false pipeline stalls but does not impact functional correctness. Furthermore, by utilizing stacked register aliasing, hardware implementations can choose to grow the number of physical registers corresponding to the stacked registers with no increase in the hazard detection logic. The increased physical registers can offset the performance loss due to false pipeline stalls by reducing the spills and fills of stacked registers.

This paper is organized as follows. The next section provides background for the register stack engine and rotating and stacked register concepts in Itanium® architecture. We also provide physical design background for hazard detection logic in Itanium® processor implementations. The third section details the concepts of stacked register aliasing and shows how the complexity of the hazard detection logic is reduced. Then, we present results of our performance simulations showing the small performance loss for integer programs.

2. Background

The Itanium® architecture defines the stacked registers as a circular set. Calls and returns move the active window, or stack frame through this set via renaming. Successive calls may move the set into a region of used registers, requiring an operation similar to a traditional register stack 'push'. Successive returns may move the set into a region of registers containing stale data from previously returned calls, requiring a stack 'pop' to restore the caller data for the next return. In order to simplify handling of the stacked registers, the architecture defines a register stack engine or RSE to implement the register stack abstraction, thus providing the software environment an unlimited number of registers by spilling or filling register data to backing store memory. Unnecessary spilling and filling of registers is avoided at procedure call and return interfaces by giving the compiler some control of the register renaming. At a procedure call interface, a new frame of registers is available to the callee, which includes the output registers of the caller. The callee can execute an *alloc* instruction to specify its intent of register usage. If sufficient registers are not available for allocation, the RSE spills registers from earlier frames to memory. This continues until the callee's request can be satisfied. Similarly upon a procedure return, the RSE may have to restore the caller's registers from memory.

Figure 1 shows an example procedure call sequence to illustrate register stacking. In this figure, the numbers to the left of the boxes represent virtual registers (VRn) and the numbers on the right represent the corresponding physical/aliased registers (PRn,Arn). It shows procedure A which has declared virtual registers 32 thru 45 as local or input registers. Virtual registers 46 thru 52 are the output registers of procedure A. Upon calling procedure B, the output registers of procedure A form the initial register frame for procedure B and are now referenced by virtual registers 32 thru 38. By performing an alloc operation, procedure B expands its register frame from virtual register 32 to 50 with virtual registers 48 thru 50 designated as output registers. This process of stacked register renaming is again repeated when procedure B calls procedure C.

In addition, the stacked registers may also participate in register rotation. Register rotation is a feature of the Itanium® architecture to efficiently support modulo scheduling of loops. The modulo scheduling of loops allows compilers to parallelize loop iterations. Register rotation is the software visible register renaming mechanism through which each iteration of a loop is provided its own set of registers.

The aforementioned characteristics of the Itanium® architecture all lead to significant demand on GR stacked register sets. This demand eventually results in increased RSE activity in order to guarantee correctness and provide software the illusion of unlimited registers. As the RSE spill and fill activities ultimately displace work-producing instructions, performance can be increased by minimizing

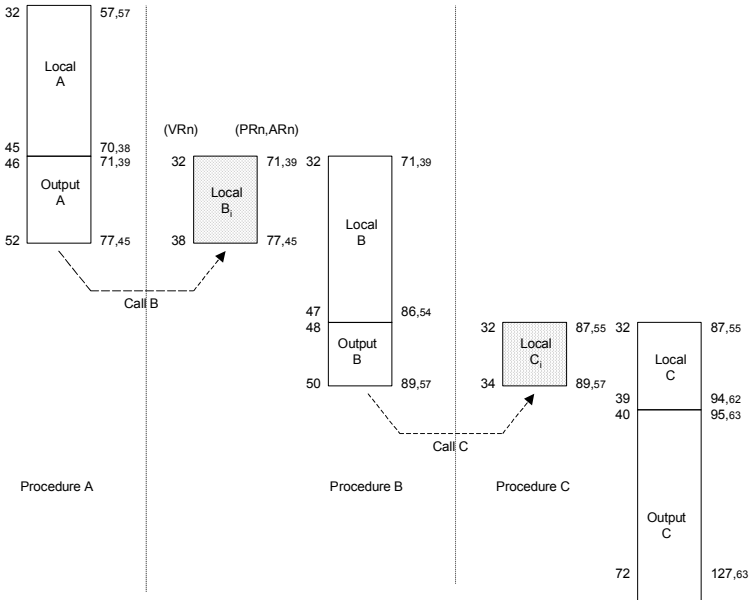


Figure 1: Example call sequence illustrating register renaming and aliasing

RSE activity. This is best accomplished in hardware by increasing the number of physical registers used to implement the stacked register sets.

Given the increasing complexity of contemporary microprocessors, new methods are required in order to limit the implementation expense of data hazard detection. Rather than traditional comparators and scoreboards for hazard detection, a new structure, which integrates both, can be used. Assume that two regids are decoded into 1-hot format; for a 128-entry GR register file a 7-bit binary-encoded regid turns into a 128-bit vector. Hazards can be then detected (in the simplest case) on a per-GR basis with a simple logical AND of the bits from each regid vector corresponding to each GR. The result is a hazard vector containing 0 asserted bits if the two regids were not equal, and a single asserted bit if the regids were equal and thus constitute a data hazard. A bitwise logical OR of the resulting hazard vector is all that is required to signal the hazard. This method would be inefficient if each compare was performed separately, but provides a method for significant hardware reduction. Since a data hazard occurs when ANY consumer regid matches the regid of ANY producer lacking a bypass path, we can combine like terms in detecting RAW hazards. By logically ORing the 1-hot vectors of all REG stage consumers, and logically ORing the 1-hot regid vectors of all older non-bypass producers, we reduce the problem (again, in the simplest case) to a single logical AND of two multi-hot regid vectors, with a bitwise OR required to generate the hazard signal. Such a structure consists of wide logical ORs and 2-input logic ANDs, and is optimal for implementation in dynamic CMOS logic. Similar methods reduce the hardware required for WAW hazard detection.

The entire hazard detection problem is, of course, more complicated than the example given, but is easily performed with multiple of these vector AND functions for different types and cases of hazards. Significant sharing of decode hardware is possible in generating the vectors for the various ANDs needed for complete hazard detection. It should be apparent that most of the 7->128 bit decodes used to generate these regid vectors are the same needed for the various read and write ports to the load scoreboards, thus re-using already required hardware. Regularity of the wiring and of the logic and decode structures further adds to the efficiency of this method of hazard detection. Each ‘row’ of the hazard logic array can be identical, and is associated with one physical register. Note that the various qualifiers can be folded into the decode logic, producing either 1-hot or all-zero vectors, depending upon the qualifier values, adding the last required layer of required complexity to the hazard detection logic.

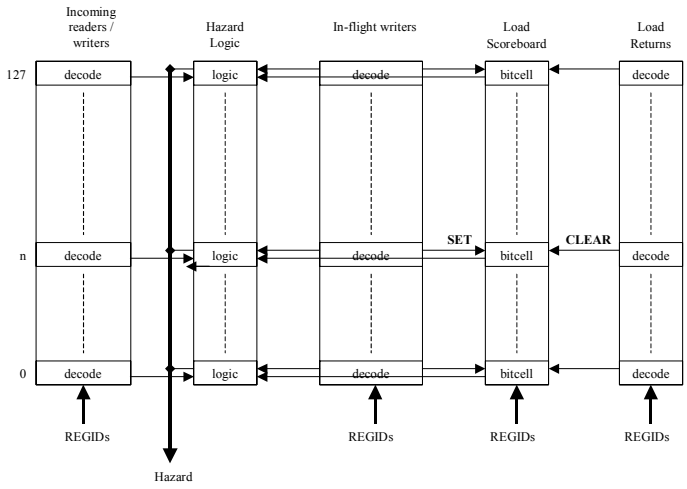


Figure 2: Hazard Logic Array Block Diagram

Figure 2 shows a block diagram of the hazard logic array physical structure. This figure shows columns of logic structures where each row is associated with a single physical register. The figure shows a 128 register high hazard logic array as an example. The far left logic column represents the source operand register identifiers of incoming or consumer instructions. These REGIDs are broadcast up the decoders and then the decoded signals travel across to the hazard logic detection column to combine with similar decode information from in-flight writers and the scoreboard information.

3. Stacked Register Aliasing

The vector-based method of hazard detection greatly simplifies the non-linear dependence of logic complexity upon the number of pipelines and pipeline stages. But it creates a linear dependence of complexity upon the number of physical registers; doubling the register file size roughly doubles the size of the hazard logic array. This dependence upon register file size might preclude increasing the number of stacked physical registers if the size of the hazard logic array grew to an unacceptable size, or if the growth slowed the logic to an unacceptable clock speed.

The size problem of the hazard detection logic may be solved by the introduction of aliasing in the hazard logic. By mapping multiple physical registers to a single row in the hazard logic, we are able to increase the number of physical registers, while maintaining or reducing the size of the hazard logic array. As an example, the 128 GRs can be represented by 64 rows in the hazard logic array. GR[127:64] would be mapped (aliased) by the decoders to the same 64 rows used for GR[63:0] in the hazard logic array. The reduction in complexity is significant, but usages of GR[127] and GR[63] are then indistinguishable. It is easy to see that a false stall can occur because of an apparent hazard caused by a REG-stage consumer of GR[63] and an older producer targeting GR[127].

In order to minimize false stalls we must consider the various types of registers used in an architecture, as well as the register frame size generally in use for those registers. We continue to have separate hazard logic arrays for GRs and FRs because their hazards are generally orthogonal – the pipeline stages and bypass structures differ considerably. In the GRs, though, recall that we have two significantly different register types; static and stacked registers. The static registers may be used at any time by any routine. The stacked GRs, however, are seen in contemporary code to be used in sets smaller than the full virtual set of 96. Even as calls and returns occur we experience a fractional window of the total to generally be referenced by the instructions in-flight at any time. Given these conditions we are best able to utilize aliasing within the stacked registers, while maintaining non-aliased hazard logic rows for the static registers.

The calling sequence shown in Figure 1 illustrates the stacked register aliasing effects. Here, 3-way aliasing i.e.

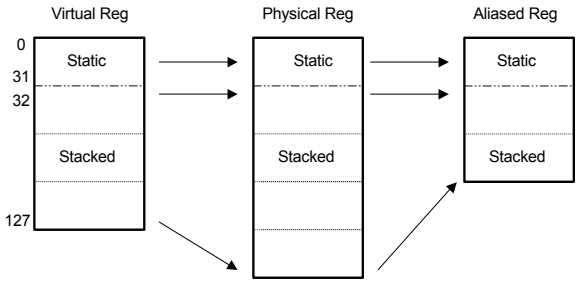


Figure 3: Register ID Mapping

the 96 stacked registers are mapped onto 32 aliased registers, is shown. Procedure B has a register frame size of 19 registers spanning from physical regid of 71 to 89. The corresponding aliased regids are from 39 to 57. Thus, this procedure will not suffer any false hazards due to aliasing. Procedure C is shown with a register frame size of 56 registers. In this case, physical regid 95 and 127 will both map to aliased regid 63. Hence, it is possible for false hazard stalls to be signaled.

If 96 physical registers are used for the 96 virtual GRs, we may represent these in the hazard logic as 48 rows (2:1 aliasing), 32 rows (3:1 aliasing), or even smaller sets, although smaller sets can easily be seen to greatly increase the detection of false data hazards. If 32 rows are used for the stacked registers, a total of 64 rows will result (32 static + 32 stacked). This eliminates 50% of the rows required with no aliasing. If only 48 rows are used, the row total increases to 80, but still represents a 37.5% reduction. These reductions are desirable even if physical register count is not increased. Reducing the number of rows has a significant effect on the power consumption of the circuit, both in the hazard logic and in the decoders. In addition, there is some reduction in circuit delays as well as the obvious reduction in area.

The benefits of aliasing are compelling as the physical register count increases. As we have discussed above, it can be desirable to increase the number of stacked physical registers in an Itanium® processor microarchitecture to reduce the number of register spills and fills. Since hazard detection is performed on physical regids, any increase in the physical registers adds complexity to the hazard logic. Hazard aliasing provides the opportunity to, at worst, minimize the increase in complexity. At best, the hazard logic can be reduced even as the number of physical registers is increased.

For a minimal change, assume that we increase from 96 to 128 the number of physical registers used for the 96 stacked GRs. We may choose the minimum (2:1) aliasing so that 64 stacked and 32 static rows are required. These 96 hazard logic rows still represent a savings of 40% verses the 160 rows otherwise required, and a 25% savings over the original solution with 96 stacked registers. These savings are not available when using explicit binary comparators for hazard detection – a 2:1 aliasing for 160 registers would offer minimal savings as 8-bit comparators simply become 7-bit comparators. Figure 3 shows how virtual registers are mapped onto physical registers and then to aliased register identifiers for hazard detection. The 32 static registers map as is all the way across on the top. The stacked registers shown in Figure 3 assume we use 128 physical registers for the 96 virtual registers and that these are mapped onto 64 hazard logic rows.

Aliasing in vector-based hazard detection is a powerful technique for mitigating the expense of hazard detection as register file sizes increase. It also adds a degree of freedom in selecting the physical size of each register file as power and area budgets can be now be dictated by the actual register file itself. But the implementation of aliasing in hazard detection obviously presents several design trade-offs. Increased aliasing ratios have the potential for performance improvements via power savings and decreased cycle time, but also increase the probabilities of false stalls. Optimizing the design requires careful performance analysis performed within the proper context, both internal and external. Internal factors are primarily related to processor microarchitecture; increased issue width and pipelined depth increases the number of stacked register frames likely to be inflight, and the memory subsystem implementation affects the number of scoreboarded loads (an important detail since these are long-latency loads and

may belong to any number of stack frames). External details are primarily related to application code and compilers, and, to a lesser degree, the operating system.

4. Performance Results

Several experiments were conducted to quantify the expected performance loss due to false hazards introduced by register aliasing. An internal performance simulator which provides a cycle accurate model for a given Itanium® processor microarchitecture was used for these experiments. The workloads were selected from Spec95 suite. Internal HP Itanium Processor Family compilers produced the binaries for these benchmarks.

Each of the benchmarks was first run without any register aliasing to collect a baseline total cycle count. Then, the performance simulator was modified to implement the register aliasing technique. Total cycle counts were collected for 2-way and 3-way aliasing.

The results of the aforementioned experiments are presented in Table 1 and 2. Table 1 shows the relative performance loss due to 2-way and 3-way aliasing for SpecINT95 benchmarks. Table 2 shows the results for SpecFP95 benchmarks.

The integer benchmarks show negligible performance loss for both 2-way and 3-way aliasing. There are a couple of reasons why this might have been expected. First, integer programs do not have as many modulo scheduled loops as technical programs, which would tend toward smaller register frames. Second, typically these programs are scheduled for a single cycle latency cache as is present in the Itanium® 2 processor. If the compiler were to schedule for higher cache latencies, the register live requirements would increase and consequently create larger register frames. The SpecFP95 benchmarks on the other hand do demonstrate some sensitivity to register aliasing. On the whole, SpecFP95 ratios declined 5.34% with 2-way aliasing and 17.48% with 3-way aliasing. One benchmark, 107.mgrid, suffered a significant increase in total cycles. The performance loss on SpecFP95 can be attributed to heavy use of modulo scheduled loops, which tend to require significantly larger register frames for the purposes of register rotation. Furthermore, SpecFP95 benchmarks were scheduled for larger cache latencies, which is on the order of 5-8 cycles.

Based on the above experimental results, one can conclude that the register aliasing technique can be implemented for the integer registers and not for the floating-point registers. Since the floating-point registers are not stacked in the Itanium® architecture, it is not necessary to provide additional physical registers beyond what is architecturally visible. However, on the integer side, the use of this aliasing technique can form the basis of a physically scalable register file implementation. For example, increasing the number of physical stacked integer registers from 96 to 128 could be accomplished with 2-way aliasing.

5. Acknowledgements

The authors would like to specially thank Terry Lyon, Greg Woods and Subramoni Parmeswaran for their assistance in the setup of performance experiments and collection of the performance data highlighted in this paper.

6. References

[1] D. Soltis, R. Bhatia and R. Arnold, “Stacked Register Aliasing in Data Hazard Detection to Reduce Circuit Size,” HP US Patent Application 10016639, Feb 2002.

[2] S. Naffziger, and G. Hammond. The Implementation of the Next Generation 64b Itanium Microprocessor. *Proc of ISSCC*, Feb 2002.

[3] E. Fetzer, and J. Orton. A Fully-Bypassed 6-Issue Integer Datapath and Register File on an Itanium Microprocessor. *Proc of ISSCC*, Feb 2002.

[4] D. Mosberger, and S. Eranian. *IA-64 Linux Kernel Design and Implementation*, Prentice Hall PTR, 2002.

[5] R. Zahir, D. Morris, J. Ross, and D. Hess. OS and Compiler Considerations in the Design of the IA-64 Architecture. *Proc of ASPLOS-IX*, Cambridge, MA, Nov 2000.

[6] Intel Corporation. *Itanium® Architecture Software Developer’s Manual*, http://developer.intel.com/design/itanium/manuals/index.htm.

Table 1: SpecINT95 Results

RegFile Alias	2-way	2-way	3-way	3-way
SpecINT95 Benchmark	**SpecINT95 Ratio** % Change	**SpecINT95 Total Cycles** % Change	**SpecINT95 Ratio** % Change	**SpecINT95 Total Cycles** % Change
099.go	0.00	0.01	-0.02	0.02
124.m88ksim	0.00	0.00	0.00	0.00
126.gcc	0.02	-0.01	-0.02	0.01
129.compress	0.00	0.00	0.00	0.00
130.li	0.00	0.00	0.00	0.00
132.ijpeg	0.00	0.08	0.00	0.00
134.perl	0.00	0.00	0.00	0.00
147.vortex	-0.08	0.03	-0.05	0.05
SpecINT95	-0.02		-0.02	

Table 2: SpecFP95 Results

RegFile Alias	2-way	2-way	3-way	3-way
SpecFP95 Benchmark	**SpecFP95 Ratio** % Change	**SpecFP95 Total Cycles** % Change	**SpecFP95 Ratio** % Change	**SpecFP95 Total Cycles** % Change
101.tomcatv	0.49	-0.49	-14.74	17.29
102.swim	0.00	0.00	-20.85	26.33
103.su2cor	-1.77	1.80	-19.82	24.71
104.hydro2d	-3.81	3.95	-13.55	15.65
107.mgrid	-30.73	44.37	-44.16	79.08
110.applu	-8.42	9.21	-13.11	15.10
125.turbo3d	-0.01	0.02	-8.48	9.26
141.apsi	-2.21	2.26	-15.70	18.62
145.fpppp	-0.32	0.30	-0.59	0.57
146.wave5	-1.66	1.69	-16.00	19.05
SpecFP95	-5.34		-17.48	

Design and Experience: Using the Intel® Itanium® 2 Processor Performance Monitoring Unit to Implement Feedback Optimizations

Youngsoo Choi Allan Knies Geetha Vedaraman Jeremiah Williamson

Itanium® Architecture and Performance Team
Intel Corporation
2200 Mission College Blvd
Santa Clara, CA
{youngsoo.choi, allan.knies, geetha.vedaraman, jeremiah.d.williamson}@intel.com

Abstract

Historically, profile-guided optimization has gathered its profile data by executing an instrumented binary and capturing the output. While this approach enables the collection of function and basic block frequencies, it cannot extract microarchitectural event information such as cache activity, TLB activity, and branch prediction behavior. Using instrumentation also requires that programs be compiled with different options (one for the profile run, one for the optimization run) with the profiling run taking substantially longer due to instrumentation overhead and reductions in compiler optimization. To help address these issues, the Intel® Itanium® 2 processor has extensive hardware support to allow for highly accurate instruction-specific information to be gathered from any binary. In this paper, we cover three broad topics: the Itanium® 2 processor performance monitoring unit (PMU), our tools and methodology to gather and process cache, TLB, and branch activity information, and a case study where we demonstrate the entire system to reduce data access stalls.

Keywords: Profile feedback, profile-guided, feedback-directed, optimization, Itanium® architecture, Intel® Itanium®2 processor, performance monitoring, compiler optimization, data prefetching

1 Introduction

Profile-guided optimization (PGO) has historically been an important technology for improving the performance of applications. On the Intel® Itanium® 2 processor, current compilers achieve a 10% to 20% performance improvement using instrumentation-based edge profiling PGO. Although PGO provides performance benefits, there are several barriers that prevent its universal usage. Most practically, PGO requires an extra step in the code development process to perform profiling runs and process the data.

Various research groups have developed systems to reduce the cost of gathering profile data [BDB00][CL99][MTB+01][PL01], but these are generally based on edge or path profiles rather than full microarchitectural monitoring. While our system still requires a second compilation step, we have dramatically reduced the overhead while increasing the amount, detail, and quality of data provided. In Section 2, we will describe the Itanium®2 processor performance monitoring features, our infrastructure for exploiting it, and measurements of overhead for gathering it.

Other systems have allowed full-time monitoring of real binaries in conjunction with re-optimization [And+97], but such approaches were generally deployed with instruction pointer (IP) sampling and extensive off-line processing to reconstruct the data into a usable form. More recent systems, such as the Alpha 21264 and Pentium 4 event based sampling systems have support for obtaining detailed microarchitectural info [Spr02] [DHW+96], but we are not aware of any current compilers that use microarchitectural feedback data. Our compiler is the first to enable program optimization based on a processor that has dedicated support for gathering detailed instruction-specific branch, cache, and TLB behavior. In Section 3, we discuss our initial experience from applying data cache access information to optimize cache behavior using an internal version of the Intel product compiler.

2 Description of the Monitors and Framework

2.1 Intel® Itanium® 2 Processor Performance Monitor Unit (PMU)

The Itanium® architecture performance monitors are designed to help characterize and optimize the behavior of applications using on- or off-line tools. The Itanium®2 PMU counters are 48-bits wide and can count as many as four microarchitectural events simultaneously (out of about 150 possible events) and allow separate or

combined monitoring of OS and application code. In addition to the basic events, the PMU provides a variety of filters (also called qualifiers) that allow finer control of counters. Available filters include an opcode matcher, instruction and data address range matchers, and a privilege level matcher. For instance, by writing the opcode pattern for loads into the PMU's opcode mask register, the instructions retired counter will only count those instructions that are loads. In general, this provides the ability to count different types of instructions using just one counter. These filtering capabilities amplify the usefulness of event counters without requiring substantially more hardware. Table 1 shows the results of using opcode matching to measure instruction mixes on the Itanium® 2 processor for the CPU2000 integer benchmarks.

	ALU	Nop	BR	FP	LD	ST	Other
164.gzip	39%	25%	10%	0%	14%	3%	9%
175.vpr	31%	24%	8%	5%	14%	5%	14%
176.gcc	38%	18%	10%	0%	14%	10%	11%
181.mcf	38%	21%	9%	0%	15%	6%	10%
186.crafty	45%	17%	9%	0%	16%	2%	11%
197.parser	36%	25%	11%	0%	15%	3%	9%
252.eon	25%	31%	4%	12%	12%	9%	8%
253.perlbmk	36%	20%	12%	0%	15%	3%	13%
254.gap	35%	19%	9%	0%	22%	5%	10%
255.vortex	39%	14%	13%	0%	15%	7%	12%
256.bzip2	40%	16%	11%	0%	15%	7%	11%
300.twolf	29%	26%	6%	5%	9%	3%	21%

Table 1 Instruction breakdown

While the Itanium®2 PMU has a large set of events and filtering capabilities, it also provides a set of *cycle accounting* events that identify and classify how time is being spent in the program. During execution, cycle accounting hardware categorizes each execution cycle into bins that represent different types of stalls. The following is a high level description of these bins:

? front-end pipeline (Icache misses)
? register stack engine (register spill/fill traffic)
? branch mispredictions and exceptions
? back-end pipeline stalls
? integer execution stalls due to register dependency
? FP execution stalls due to register dependency
? Load to use stalls (on cache misses)

Any cycles not binned into one of these categories are 'unstalled cycles' and represent times when the pipeline is flowing freely. Cycle accounting is organized hierarchically so a user can do a high level binning with the top-level categories and then drill down into specific trouble areas by fine-tuning the event parameters. Figure 1 shows the cycle accounting breakdown for the Itanium® 2 processor running the CPU2000 integer benchmarks.

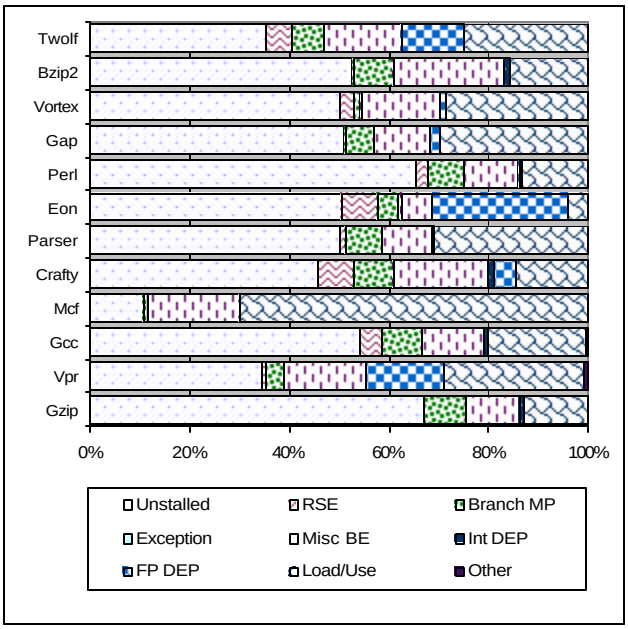


Figure 1 Cycle accounting for CPU2k Integer

While the features described so far are very valuable in characterizing the overall behavior of applications, for feedback optimization, we are interested in event data that is linked to specific locations in the program. To do this, the Itanium architecture provides event address registers (EARs) that can be programmed to capture microarchitectural event information and the instruction pointer (IP) which caused the event. There are EARs for data, instruction, and branch events. Each of these EARs captures the IP plus EAR-specific event information.

The data EAR can capture events related to cache, TLB, and ALAT misses. When programmed to track data cache misses, the processor samples load misses at a programmable frequency and records the instruction address, the data address, and the latency needed to access the data. When programmed to monitor TLB activity, the data EAR captures the level in the hierarchy that services the translation (first or second level TLB, hardware page walker, or the operating system). When programmed to capture ALAT information, only the IP of chk.a or ld.c that miss in the ALAT are recorded.

For branches, the 'EAR' is actually an eight entry branch trace buffer (BtrB) that continuously captures the IP and target address or prediction information of branches. The BtrB can be programmed to record all branches or specific subsets of branches (only taken or not-taken branches, only branches that correctly or incorrectly predict the target or direction, or only for

branches that are of specific types such as call, return, loop-type, etc.). The BtrB continuously gathers events until a programmed threshold of collected events is reached (e.g., X mispredicts, or Y branches executed, etc.)

Using these capabilities, the BtrB can be used to replace instrumentation-based edge profiling with statistical edge/block profiling by sampling the BtrB and mathematically reconstructing approximate edge weights for the graph. Other groups at Intel have implemented such systems [BN00] and have demonstrated that the statistical sampling of the BtrB yielded profiles comparable to instrumentation-based approaches.

The BtrB can also be used to identify the branches that mispredict most frequently by programming the buffer to only record those branches that mispredict. The mispredicted branches are then statistically sampled from the BtrB. This section has provided just a brief introduction to the capabilities of the Itanium® 2 PMU – please see [Int02] for full details.

2.2 Programming, Sampling, and Storing PMUs

To program and control the PMU, we use a tool called *psamp*, which is part of Intel's Vtune™ Performance Analyzer[1][IntV02]. *Psamp* takes command line inputs and writes the performance monitor control (PMC) registers to set the sampling rates, filtering options, events to count, etc. *Psamp* gathers EAR samples in an internal buffer until it fills up or the monitored application terminates, at which point the samples are written to disk in a binary format called .TB3.

The overhead for collecting the EAR samples is relatively small and configurable. Table 2 shows the overhead for enabling sampling using the SPEC CPU2000 integer benchmarks as the workload and measuring the impact of different sampling rates.

Event on which sampling is based	Sampling rate					
	1/1M	1/500k	1/100k	1/50k	1/10k	1/1k
DATA_EAR_EVENTS (Cache Miss)			0.19%	0.28%	0.89%	5.33%
DATA_EAR_EVENTS (TLB Miss)			0.14%	0.14%	0.27%	1.53%
BRANCH_EVENT	0.44%	0.86%	3.45%	6.85%		
CPU_CYCLES	1.00%	2.23%	8.80%	15.98%		

Table 2 Sampling overhead as a percentage of total execution time

The first three rows show the overhead for sampling based on the DATA_EAR_EVENTS and BRANCH_EVENT events (which count the number of data EAR events and BtrB events respectively). The fourth row shows the overhead for sampling the IP when the CPU_CYCLES event counter overflows. For BRANCH_EVENT and CPU_CYCLES, sampling every 500K events incurs only 1%-2% overhead. For data cache misses, we found that sampling 1 in 50K misses successfully identified hot loads just as well as sampling more frequently while incurring less than 1% overhead.

Since the .TB3 file contains all the samples taken during a monitoring run, we use a post-processing utility called *sfdump* to extract and summarize the necessary information into a text format called the MDB file (Monitor Data Base). In the MDB file, all samples with the same IP are stored in a single combined entry resulting in a 10-100x reduction in file size (versus the .TB3 file).

2.3 The Annotations Library and a Compiler Usage Methodology

Since all the data gathered by the EARs is indexed by IP, the compiler needs a mechanism to map instructions in its internal intermediate representation to the event data saved in the MDB file. To do this, the compiler uses an annotation system to save information such as program control flow graphs, basic block attributes, and instruction level line/column numbers into the binary (much like debug information). Later, the compiler uses an annotations library to read the annotations back out of the binary (by IP) and maps it to PMU data from the MDB file. The annotations library is open source and available for download [Int01].

Although the annotations library can accommodate nearly any structural organization that a compiler might want to emit, we have chosen a convention that allows tools to traverse the annotation hierarchy without knowing all the possible attributes that might be contained in a given binary. These conventions allow new annotations to be emitted by a compiler without having to rewrite existing tools (forward and backward compatibility). The conventions define a program hierarchy of module – file – function – basic block – instruction. At each level, the compiler describes what attributes hang off of each object as well as the parent/child relationship with the other levels (e.g., basic blocks are children of functions).

To build the entire annotation structure, the compiler recursively traverses its internal program representation and creates corresponding objects in the annotation hierarchy and fills in their attribute fields. Since it is likely that not every attribute of every object needs to be given a value, the annotations library optimizes space by compressing unused attribute fields. Once the compiler has completed annotating the binary, they are emitted into the assembly file. Although the annotations increase the physical size of the binary, the linker is directed to store annotation information into a separate section in the binary so that it will not be loaded at run-time.

[1] Similar tools are available for other platforms, such as pfmon for Linux and Caliper for HPUX.

Figure 2 shows the conceptual flow of an annotation-based performance analysis infrastructure. Following the middle column, an application is compiled with the annotations library and the resulting binary is run on hardware with *psamp* gathering PMU data. The resulting .TB3 file is then converted to MDB format. At that point, we have two possible usage models. In both flows, the annotated binary is read and the annotations converted to a text format that is easy to read/parse.

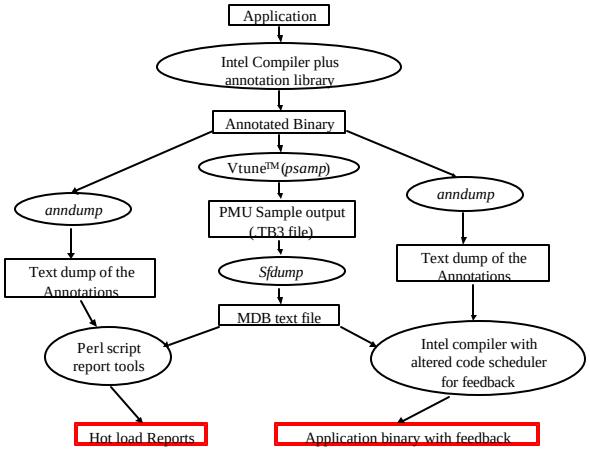


Figure 2 Conceptual annotation based infrastructure

In the left flow, we use the MDB file and the text version of the annotations extracted from the binary to feed an analysis tool that creates run-time reports. By combining the annotation and PMU data, these reports can show what events happened and where (function, block, instruction). Since the MDB and annotations are both available in text format, the tools can be written in a shell language such as Perl.

In the right side flow, the annotated binary is again converted to text, but this time, the MDB and annotation text files are fed back into the compiler, which recompiles the application using the PMU data to enhance its heuristics.

For both flows in Figure 2, Figure 3 shows the specifics of how the MDB and annotations data correlate the dynamic run-time behavior with the static compile-time information. Since the annotations contain the function name, source line and column number, and the IP in the binary, we are able to map the IP event data to an instruction's intermediate representation in the compiler or just use them to print a runtime report for performance analysis.

The infrastructure can handle any IP-specific event data that are generated by the Itanium®2 PMU, although we specifically focused on the data EAR for our case study in the next section.

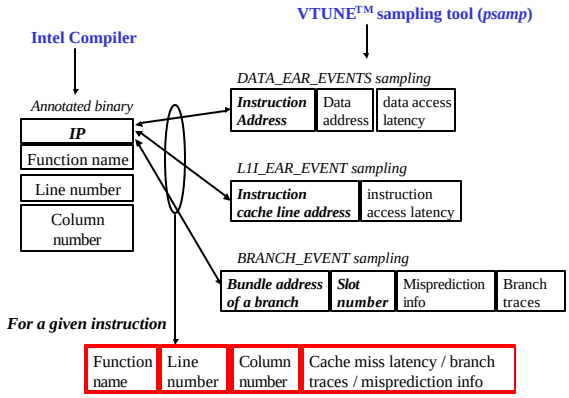


Figure 3 Correlating compile-time information with PMU results

3 Case Study: Reducing Memory Access Stalls

To test our infrastructure, we wanted to attack a problem that has not been effectively handled using current production techniques. Since memory access stalls are one of the largest components of runtime in many applications, we wanted to use PMU features to help reduce this component. Although the compiler already emits explicit prefetches for some predictable data access patterns (e.g., array or linked-list access with fixed strides), it generally doesn't know which loads are missing in cache when it decides to use a prefetch. The fact that we can easily gather this information in the PMU motivated this study more than having specific new prefetching heuristics/ideas in mind.

Additionally, we wanted to demonstrate results on a real system with all their practical limitations and restrictions. Unfortunately, unlike a project where one is experimenting with 'what if' scenarios (what if the processor PMU had this capability, what if the compiler didn't have this type of limitations, etc.) this made it very difficult to try ideas that depending on fixing new-found limitations of our system. On the other hand, all the results shown *demonstrate* lower bounds on what can be achieved. The process of implementing the infrastructure on near-production systems also provides practical learnings on what will be required to productize our techniques. In Section 3.2, we describe our heuristics. In Section 3.3, we analyze the results of applying these heuristics and running the resulting binaries on hardware. In Section 3.4, we discuss the high level learnings from the case study.

3.1 Experimental Setup

Our experiments were run on a single pre-production Intel® Itanium®2 processor using the Intel 870 chipset and 1 GB of main memory. The Itanium®2 processor can issue a maximum of six instructions per clock, composed of up to two integer loads, two stores, six arithmetic operations, three branches, and two floating-point operations. The first level instruction and data caches are 16 KB each, 4-way set associative. The second level cache is 256KB unified, 8-way set associative and the third level cache is 3MB, 12-way set associative. A complete description of the Itanium® 2 processor micro-architecture is available in [Int02].

For our experiments, we selected the SPEC CPU2000 integer benchmarks[Spe00] and a few benchmarks from the Olden benchmark suite[Old96] that are known to be data intensive. The baseline executables were built with instrumentation-based feedback, O2 optimization, and whole-program interprocedural analysis using an internal version of Intel's product Itanium® architecture compiler [IntC02]. [2]

Since the compiler's global code scheduler is a complex decision engine, the only changes we could make that maintained the compiler's scheduling infrastructure were to adjust the scheduling priorities of various instructions, insert new prefetch sequences, and adjust the decision logic that determined whether to speculate a particular instruction. After those choices were made, we let the scheduler continue on its normal path (e.g., considering resource contention, compensation code costs, dependence heights, etc.) to maintain high code quality. Since the Intel compiler has a separate scheduler for software pipelined loops, we turned off software pipelining so that all loop bodies were scheduled using the normal global code scheduler plus our heuristics (turning off software pipelining incurs about a 1% loss on CPU2000 integer codes).

3.2 Feedback Heuristics

We implemented six different scheduling heuristics, each of which can be put into one of two categories, those that insert explicit data prefetch instructions (*lfetch*) and those that try to increase the distance between loads and their dependent instructions. The former helps reduce data cache misses by fetching data in advance of the target load instruction, while the latter attempts to delay use of loaded values that are likely to miss in cache. Table 3 provides a summary of the transformations and scheduling heuristics. The first three rows are the prefetching techniques (H_NEXP, H_GENP, H_BADP), the next 2 rows are scheduling techniques (H_UMUS, H_MAXD), and the last row is a combination of heuristics (H_COMP).

H_NEXP Next line prefetching	To take advantage of spacial locality, prefetch the next 128-byte second-level cache line.
H_GENP General prefetching	To exploit general prefetching, insert a prefetch to the same address as the load, but give the prefetch the highest scheduling priority in the compiler.
H_BADP Base address prefetching	To compute the address of an upcoming load early, take advantage of loads whose addresses are computed as base+constant. Insert a prefetch to the base address and give it very high priority in the scheduler.
H_UMUS Use miss unspeculation	To minimize the stalls related to speculative loads, do not hoist uses of hot loads from their home block. This avoids unnecessary cache miss stalls due to speculative computations whose results might not be needed.
H_MAXD Max load-use distance	To increase the distance between a load and its dependent operations, assign the load a very high priority and dependent instructions a very low priority.
H_COMP Combination prefetching	To combine a few of the heuristics, the compiler selectively applies the combination of the heuristic *H_NEXP, H_BADP, and H_UMUS* depending on the average latency of each load's data cache misses. Temporal locality completers are used to prevent the first level cache from being polluted when the average latency of a load in the *H_NEXP* heuristic is more than 22 cycles.

Table 3 Description of the heuristics and transformations for hot loads

In order to concentrate our heuristics on important loads, we identify those loads that accounted for a significant (either 2 or 6) percent of a program's overall cache cycles as being *hot loads.* For H_NEXP, the threshold was 6%; for all other heuristics, it was 2%. These thresholds were determined by performing test runs to see which values provided the best average performance, although we have not fully investigated complete combinations or individual thresholds for each benchmark.

To implement these heuristics, we altered the compiler's global code scheduler to recognize which loads are hot loads as they became candidates to be scheduled. At that point, we altered either the scheduling priorities (for H_UMUS and H_MAXD) or inserted prefetches (for H_NEXP, H_GENP, H_BADP).

As shown in the example in Figure 4, next line prefetching (H_NEXP) exploits spatial locality for hot loads by prefetching the *next* sequential cache line. The line sizes of Itanium® 2's first and second-level data caches are 64 bytes and 128 bytes, respectively. We chose to prefetch for the second level cache line size since the cache is only five cycles away and experimental results confirmed that it provided better performance on average than prefetching first level cache lines.

[2] Note: the baseline for the Olden benchmarks were not compiled with PGO. Although this impacts the results, our experience is that the percentage of execution time due to data cache stalls is not greatly affected by the optimization level of our current compilers (except for MCF where prefetching is used extensively at high optimization levels).

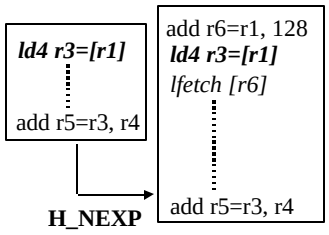


Figure 4 Transformation by heuristic H_NEXP

To try to take advantage of addresses that are available early, general prefetching (H_GENP) inserts lfetches for hot loads and gives the lfetch the highest possible priority for scheduling. Figure 5 shows this transformation. Since both load and lfetch have the same address in this heuristic, the lfetch's scheduling is also bounded by address computation. However, since an lfetch does not change register values and is independent of control and data dependencies, it can be hoisted across calls, branches, and stores without any special support.

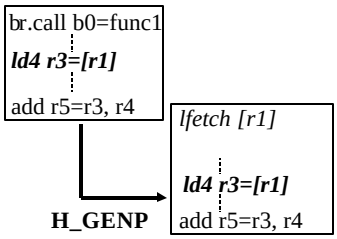


Figure 5 Transformation by heuristic H_GENP

Base address prefetching (H_BADP) searches for loads whose addresses were computed as base+small constant offset and inserts an lfetch for the base address. Figure 6 illustrates this transformation. Since the Itanium architecture does not provide base+offset addressing for memory operations, this transformation guarantees that the prefetch can be scheduled at least one cycle earlier than the load being prefetched.

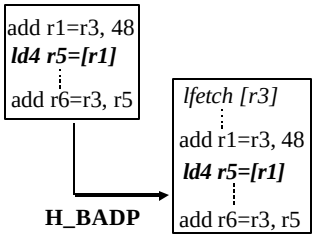


Figure 6 Transformation by heuristic H_BADP

While the first three heuristics tried to improve cache access times by inserting prefetches, the next two heuristics change the way the compiler schedules hot loads and their consumers, but does not insert any prefetch instructions. Use miss unspeculation (H_UMUS) identifies hot speculative loads and attempts to prevent their consumer instructions from being speculated. Figure 7 shows the code before and after speculation. Notice that if the speculated load misses in cache, the *add* instruction will stall waiting for the data, independently of whether the program executes the path from which they were hoisted. For example, if the load misses in cache and then the branch to *label4* were taken, we would have suffered an unnecessary cache-miss-use stall since the results of the load and add are never used.

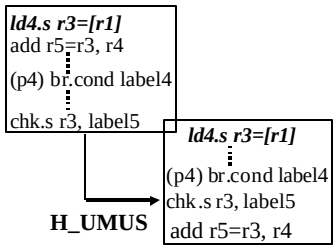


Figure 7 Transformation by heuristic H_UMUS

By forcing the consumers of hot speculative loads to stay in their home blocks, we both increase the distance between a producer and consumer, and we eliminate unnecessary cache-miss use stalls.

As shown in Figure 8, maximize-distance scheduling (H_MAXD) tries to maximize the distance between loads and their consumer instructions. The compiler computes scheduling priority for each instruction and chooses one that has highest priority. By giving highest priority to the loads that miss the data cache and the lowest priority to their consumer instructions, any slack available in the schedule is used to increase the distance between hot loads and their uses. In addition, to further increase distance, we changed the expected latency of hot loads to 10 cycles (the maximum latency to access second-level data cache).

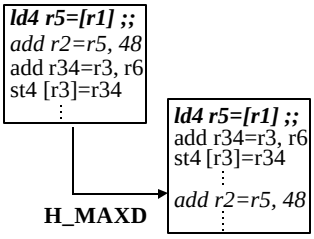


Figure 8 Transformation by heuristic H_MAXD

Finally, to take our learnings from the other approaches, we created a combined heuristic (H_COMP). In this heuristic, we also took advantage of the latency

information provided by the data EAR (on average, how many cycles did each load take to return data) in addition to the basic thresholds described earlier. For H_COMP, we apply H_UMUS in all cases. Additionally, H_COMP tries to apply H_BADP for the loads that miss the data cache and that took 7 or more cycles to return data (experiments showed those with shorter latencies did not benefit). If a given hot load did not qualify for that transformation, then the compiler attempted to apply H_NEXTP if the load showed more than a 12 cycle latency (to bring data in to the second-level cache).

For all of our heuristics, we made several compromises for the sake of completing the case study using available hardware and software. First, we only work on hot loads without respect to their context (cyclic or acyclic regions). This means that our experiments were not able to prefetch across loop iterations. Second, we did not have access to information about address patterns from the high level optimizer to help us make more intelligent choices. Third, we applied the definition of hot loads uniformly, thus ignoring potential benefit from considering criticality and the potential impact of resource constraints. To help counterbalance the resource/criticality issue, we inserted a post-scheduling clean-up phase that removed lfetches if they were scheduled less than two cycles before the load they were targeting. This helps to reduce pressure on the memory system from lfetches with limited potential to reduce latency.

3.3 Results

Figure 9 shows results from SPEC CPU2000 integer benchmarks. Each bar represents the ratio of a heuristic that uses PMU feedback versus the baseline binaries (which are heavily optimized with instrumentation-based feedback, whole program optimization, but no use of PMU data). The first cluster of bars shows relative CPU cycles, the second cluster shows the amount of time spent waiting for loaded data, and the third cluster, the number of first level cache misses.

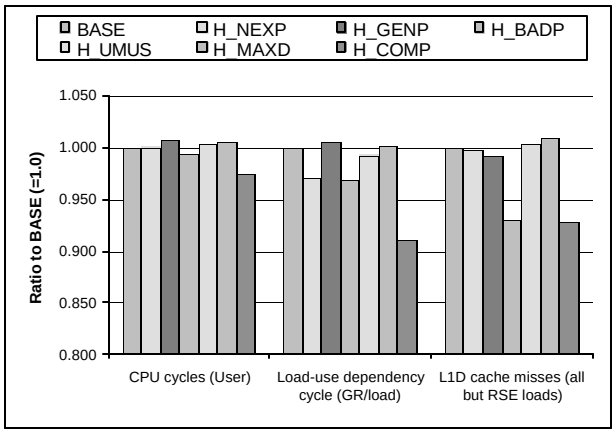


Figure 9 Average results from SPEC CPU2000 integer benchmarks

The combined heuristic *H_COMP* shows the best average speedup (2.5%) with most of the gain coming from mcf (9.4%) and bzip2 (8.1%). H_COMP also reduces average first-level data cache misses by 7% and the average amount of time that instructions are stalled waiting for cache misses by 9%. Next line prefetching (*H_NEXP*) worked especially well for gap (3.9%) and base address prefetching (*H_BADP*) was effective for mcf (9.7%) and bzip2 (2.5%), but the impact on the other benchmarks was not significant.

H_GENP, *H_UMUS*, and *H_MAXD* did not generally improve performance. For H_GENP, the compiler ended up scheduling lfetches too close to the load they were intended to prefetch and were later removed by the cleanup phase. Since H_UMUS and H_MAXD attempt to create distance between a load and its use, this generally results in uses being delayed. While this is beneficial when the load misses in cache, it is harmful when the load hits in cache (and the use could have executed earlier). Thus, we found that these heuristics are only affective for a few spot loads in gzip where the loads miss so frequently that the benefit during cache misses outweighs the losses during cache hits.

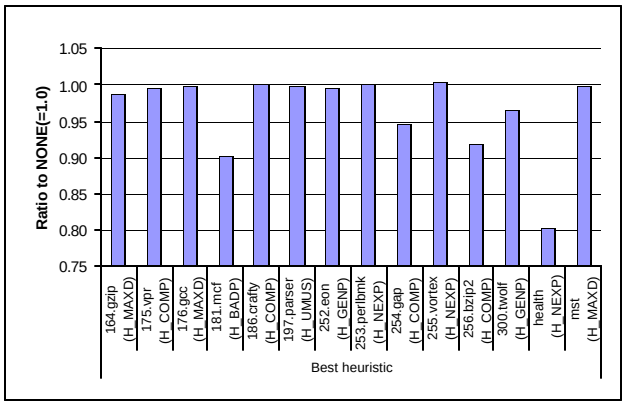


Figure 10 CPU cycle changes by best heuristic for each benchmarks

Figure 10 shows changes in CPU cycles by best heuristic for each benchmark. Each bar is tagged by a benchmark name and the heuristic that worked best for that benchmark. This graph shows that a few benchmarks benefited substantially with our heuristics, but that no one heuristic worked across the board.

Health showed a particularly interesting behavior, where basic next cache line prefetching (*H_NEXP*) reduces CPU cycles by 20%. With further investigation, we found that the single hottest load in the program missed the second-level cache 50% of the time. To avoid perturbing the first level cache, we hand-appended the *.nta* cache placement hint (no temporal locality all levels of the cache) to the lfetch. The .nta hint causes the processor to only bring the line into the second-level cache and to mark it as next to be replaced. This yielded a 46% speed-up due to avoiding cache pollution by unused prefetches. The realization that small changes in heuristics caused great volatility in response was what led to our creating the combined H_COMP heuristic.

Since the results of the heuristics were so difficult to interpret (no consistent winners, small changes making large differences in performance, large changes making little or no difference in performance), we performed an extensive measurement of how well the lfetches that were inserted 'covered' the loads that missed in cache.

Column A	Total number of sampled data cache misses
Column B	The number of cache misses for which the compiler inserted lfetches
Column C	The number of Column B's prefetches were later removed by the compiler
Column D	The final number of cache misses actually covered by remaining lfetches
Column E	The data cache miss coverage (%) by prefetch instructions (= Column D / Column A)

Heuristic	Benchmark	A	B	C	D	E
H_NEXP	164.gzip	25,623	276	0	276	1.08%
H_GENP		25,623	13,666	11,482	2,184	8.52%
H_BADP		25,623	0	0	0	0.00%
H_COMP		25,623	276	0	276	1.08%
H_NEXP	175.vpr	31,093	196	0	196	0.63%
H_GENP		31,093	20,025	15,150	4,875	15.68%
H_BADP		31,093	17,346	573	16,773	53.94%
H_COMP		31,093	196	0	196	0.63%
H_NEXP	176.gcc	4,430	24	0	24	0.54%
H_GENP		4,430	1,360	1,269	91	2.05%
H_BADP		4,430	391	144	247	5.58%
H_COMP		4,430	24	0	24	0.54%
H_NEXP	181.mcf	22,507	14,210	0	14,210	63.14%
H_GENP		22,507	18,554	15,258	3,296	14.64%
H_BADP		22,507	12,846	2,167	10,679	47.45%
H_COMP		22,507	17,626	0	17,626	78.31%
H_NEXP	186.crafty	9,529	0	0	0	0.00%
H_GENP		9,529	226	226	0	0.00%
H_BADP		9,529	0	0	0	0.00%
H_COMP		9,529	0	0	0	0.00%
H_NEXP	197.parser	23,197	0	0	0	0.00%
H_GENP		23,197	8,194	6,559	1,635	7.05%
H_BADP		23,197	4,228	978	3,250	14.01%
H_COMP		23,197	0	0	0	0.00%
H_NEXP	252.eon	33,778	46	0	46	0.14%
H_GENP		33,778	497	497	0	0.00%
H_BADP		33,778	109	60	49	0.15%
H_COMP		33,778	46	0	46	0.14%
H_NEXP	253.perlbmk	7,524	111	0	111	1.48%
H_GENP		7,524	2,219	2,173	46	0.61%
H_BADP		7,524	1,594	67	1,527	20.30%
H_COMP		7,524	1,423	0	1,423	18.91%
H_NEXP	254.gap	4,307	1,479	0	1,479	34.34%
H_GENP		4,307	2,812	1,298	1,514	35.15%
H_BADP		4,307	845	27	818	18.99%
H_COMP		4,307	1,479	0	1,479	34.34%
H_NEXP	255.vortex	8,001	24	0	24	0.30%
H_GENP		8,001	4,272	3,466	806	10.07%
H_BADP		8,001	1,830	741	1,089	13.61%
H_COMP		8,001	638	0	638	7.97%
H_NEXP	256.bzip2	18,142	1,780	0	1,780	9.81%
H_GENP		18,142	16,504	16,295	209	1.15%
H_BADP		18,142	10,941	0	10,941	60.31%
H_COMP		18,142	12,721	0	12,721	70.12%
H_NEXP	300.twolf	36,329	4,852	0	4,852	13.36%
H_GENP		36,329	12,275	7,564	4,711	12.97%
H_BADP		36,329	3,794	797	2,997	8.25%
H_COMP		36,329	4,852	0	4,852	13.36%
H_NEXP	Sum	224,460	22,998	0	22,998	10.25%
H_GENP		224,460	100,604	82,043	18,561	8.27%
H_BADP		224,460	53,924	4,909	49,015	21.84%
H_COMP		224,460	39,281	0	39,281	17.50%

Table 4 Data cache miss coverage by *lfetches* per benchmark

In Table 4, Column A shows the total number data cache miss samples were gathered, and Column B the number for which we inserted lfetches. Column C shows how many were removed because they were eventually scheduled in the same cycle as their target load and Columns D and E show the number and percent of misses that were finally covered at runtime by the inserted lfetch instructions. Note that 'coverage' implies that we

successfully inserted a prefetch, but does not indicate how much of the latency was covered.

Although the heuristic *H_COMP* was our best performer, it covered only 17.5% of total data cache misses. In the case of heuristic *H_GENP*, the initial lfetches inserted covered almost 50% of total data cache misses. However, after the clean-up phase, only 8% of misses were still covered. These results indicate that our detection mechanism (the data EARs, sampling levels, and thresholds used to classify loads as hot or not) are effective at identifying important loads, but that our scheduling/prefetching algorithms are not sufficiently general to handle all the situations.

3.4 Discussion

Although we had some success inserting prefetches, we found that many loads were difficult to optimize because of three basic problems: they were in a short code sequences (e.g., tight loop bodies), they were closely paired with their dependent instructions, or their address computation was on the critical path and lfetches could not be inserted early enough to be effective. The learning is that future efforts need to enable feedback to the loop optimizer to handle cross-iteration scheduling and software pipelining to increase miss coverage.

Our second major learning was that prefetching for locality is easier than prefetching for latency (even with hot load information). We found that we could effectively prefetch second-level cache misses with H_NEXP, but prefetching for loads that hit in the first or second-level cache were far less effective. Part of this is due to the limitations of our scheduler, but part of it is also the inherent criticality of address computations.

The third learning came from the product compiler team's experience and relates to our results. On early steppings of the Itanium® 2, the product compiler team had difficulty effectively using prefetches on floating-point codes. It was discovered that there were limitations in the processor's ability to handle overlapping (usually redundant) outstanding prefetches. When later processor steppings came out that improved the processors' ability to handle overlapping prefetches, performance of the product compiler's prefetching improved dramatically. This shows that relatively small differences in prefetching hardware greatly affect software's ability to use the feature. In this case, the hardware's ability to dispose of overlapping prefetch instructions dramatically improved the usability of the feature for the compiler. The results from our experiments were performed on early steppings of the Itanium® 2 processor and may thus have further room for improvement.

The last major learning is that everything has to be designed to work together from the ground up. We saw that no one heuristic can be used for all benchmarks or even within an individual benchmark. We also found that the scheduler and high-level analysis components of the compiler all need to be cooperating and prepared to use the information that our PMU infrastructure provides. Finally, we saw that even with the hardware prefetch instructions, seemingly small behavioral characteristics greatly impacted the compiler's ability to use prefetches.

4 Related Work

To increase program performance, compilers, binary optimizers and other tools have incorporated dynamic information to bridge the gap between global static information in the compiler and local dynamic information from the processor. In [Smi00], Smith provides a comprehensive review of the challenges present in utilizing feedback-directed optimizations.

Traditionally, tools such as compilers, profilers, and post-link binary optimizers [LS95][Smi91][Rat98][SE94] instrument the code to create a dynamic profile that is then fed to an extra round of optimization [BL94][BL00][CL99][Hwu+93]. Some approaches use hot path information to optimize code layout [DB00]. Spike [CGL+97] uses sampled IP data to reconstruct hot paths and reorders code to improve instruction cache performance. Morph [ZWG+97] is an OS supported profile-directed optimization framework to seamlessly collect and maintain profiles that are used to transform an intermediate representation of the program into a new executable. Dynamo [BDB00] is a software-based on-the-fly dynamic optimization engine used to transparently gather profile information and optimize HP PA-8000 binaries.

With the advent of performance counters maintained by hardware, microarchitectural effects such as caching and branch prediction that were once hidden from external tools can now be used to increase performance. ProfileMe [DHW+96] samples instructions and provides highly detailed information by following individual instructions through the entire pipeline. Event-based sampling in the Pentium 4 [Spr02] allows events to be tracked to exact IPs. Other approaches have tried to adapt existing structures to provide profile data [CPC94].

Focusing on the desire to integrate profile collection and optimization, entire dynamic optimization frameworks have been proposed and evaluated. DCPI [And+97] is a low overhead hardware-based continuous profiling system that samples program counters. The system buffers multiple samples in the hardware using hash tables before invoking an interrupt, thus amortizing the cost of the interrupt over multiple samples. The Compaq C compiler [CL99] also performs many code layout optimizations based on data collected with DCPI. Similar to our experimental framework, HP Caliper [Hun00] takes advantage of the Itanium®2 PMU to analyze applications' performance and provides user APIs for dynamic as well as static binary instrumentation.

The desire to increase communication between static environments and hardware has led to proposals for dynamic optimization in hardware. Replay [PL01] and Daisy [EA97] both provide a microarchitectural framework to support dynamic optimization. The hardware is given the role of constructing hot code sequences, performing optimizations, and recovering from errors. Merten et. al. [MTB+01] propose another such framework based on the IMPACT [ACM+98] compiler.

5 Conclusions

In this paper, we have demonstrated the ability to enhance instrumentation-oriented feedback approaches with hardware-based solutions by enhancing existing infrastructure to make use of new hardware and software usage models. Together with results from previous studies, our work shows that PMU-based statistical sampling can completely eliminate the need for code instrumentation while simultaneously increasing ease of use, speed of profiling, and the level of information that can be gathered. To make this possible, performance monitoring hardware such as that provided by the Itanium® 2 processor event address registers must be available and reasonably efficient.

In our case study, we learned that having the information is not the same as using the information. In our data cache study, we were able to improve scheduling heuristics and insert prefetch instructions targeted at loads known to miss in cache. The results showed that individual heuristics were effective at improving specific benchmarks, but that no one heuristic worked across all benchmarks or even all locations within a single benchmark. These results demonstrated that the entire optimization framework needs to be well-engineered with great attention to every component to effectively make use of the data – hardware, driver, scheduler, software pipeliner, and high-level optimizer.

While our study only concentrated on data cache access, our infrastructure supports the feedback of any IP information the Itanium® 2 processor can deliver, including instruction stream, branch prediction, TLB, and ALAT event information. Now that the infrastructure for testing new compilation heuristics is available, we expect this information to drive the next generation of optimizations to make better use of the many advanced features in the Itanium architecture.

While this paper was targeted at static compilation, efforts are already underway to leverage these techniques for use in managed runtime systems such as Java or .NET. In these systems, low-overhead, high-detail profiling will allow for universal recompilation/optimization of application codes universal while being invisible to the end user. As an intermediate step, another possibility is to productize 'self-monitoring' software that can self-select different code sequences based on real-time PMU feedback. The Itanium®2 processor's PMU and the work presented in this paper are just the beginning steps in enabling the continuous low-overhead collection of detailed microarchitectural behavior to improve application performance.

6 References

[ACM+98] D.I. August, D.A. Connors, S.A. Mahlke, J.W. Sias, K.M. Crozier, B. Cheng, P.R. Eaton, Q.B. Olaniran, and W.W. Hwu. "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture," *Proc. 25th Int'l Symp. on Computer Architecture*, June 1998.

[And+97] J. Anderson, et al. "Continuous Profiling: Where have all the cycles gone?," *Proc. 16th ACM Symposium on Operating Systems Principles*, October 1997.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. "Dynamo: A Transparent Runtime Optimization System," *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[BL94] T. Ball and J. Larus. "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems, Vol. 16, No. 4*, July 1994.

[BL00] T. Ball and J. Larus. "Efficient Path Profiling," *Proc. 29th Annual Intl. Symp. on Microarchitecture*, June 2000.

[BN00] J. Bharadwaj and R. Narayanaswamy. "Continuous Compilation," *Unpublished Report*, October 2000.

[CGL+97] R. Cohn, D. Goodwin, G. Lowney, and N. Rubin. "Spike: An Optimizer for Alpha/NT Executables," *USENIX Windows NT Workshop*, August 1997.

[CL99] R.Cohn and G. Lowney. "Feedback Directed Optimization in Compaq's Compilation Tools for Alpha," *Proc. 2nd ACM Workshop on Feedback-Directed Optimization*, November 1999.

[CPC94] T. M. Conte, B. A. Patel, J. S. Cox. "Using Branch Handling Hardware to Support Profile-Driven Optimization," *Proc. 27th Annual Intl. Symp. On Microachitecture*, November 1994.

[DB00] E. Duesterwald and V. Bala. "Software Profiling for Hot Path Prediction: Less is More," *Proc. 9th Annual Intl. Conf. On Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[DHW+96] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. 30th Annual Intl. Symp. on Microarchitecture*, December 1997.

[EA97] K. Ebocioglu and E. Altman. "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proc. 24th Annual Intl. Symp. on Computer Architecture*, June 1997.

[EAG+01] K. Ebocioglu, E. Altman, M. Gschwind, and S. Sathaye. "Dynamic Binary Translation and Optimization," *IEEE Trans. On Computers, Vol. 50,*

No. 6, June 2001.

[Hun00] R. Hundt. "HP Caliper: A Framework for Performance Analysis Tools," *IEEE Concurrency, Vol. 8, No. 4*, October-December 2000.

[Hwu+93] W.W. Hwu et. al. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, January 1993.

[Int01] Intel Corporation. Flexible Annotations API Programmer's Guide, May 2001, *Available from http://developer.intel.com/software/products/opensource/tools1/perftools.htm*.

[Int02] Intel Corporation. Intel® Itanium®2 Processor Reference Manual for Software Development and Optimization. June 2002. *Available from http://developer.intel.com/design/itanium2/manuals/index.htm*.

[IntC02] Intel Corporation. Intel Performance Compilers, *Available from http://developer.intel.com/software/products/compilers*.

[IntV02] Intel Corporation. Intel Vtune™ Performance Analyzer, *Available from http://developer.intel.com/software/products/vtune*.

[LS95] J. Larus and E. Schnarr, "EEL: Machine Independent Executable." In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, ACM, pages 291-300, June 1995.

[MTB+01] M.C. Merten, A.R. Trick, R.D. Barnes, E.M. Nystrom, C.N. George, J.C. Gyllenhaal, and W.W. Hwu. "An Architecture Framework for Runtime Optimization," *IEEE Trans. On Computers, Vol. 50, No. 6*, June 2001.

[Old96] The Olden benchmark suite, June 1996, *Available from http://www.cs.princeton.edu/~mcc/olden.html.*

[PL01] S.J. Patel and S.S. Lumetta. "rePLay: A Hardware Framework for Dynamic Optimization," *IEEE Trans. On Computers, Vol. 50, No. 6*, June 2001.

[Rat98] Rational Software Corporation. Purify, 1998. http://www.pure.com/products/purify.

[SE94] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools." In Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, ACM, pages 196-205, June 1994. See also Research Report 94/2, Western Research Laboratory, Digital Equipment Corporation.

[Smi91] M. Smith, "Tracing with Pixie." Technical Report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, Stanford, CA, USA, November 1991.

[Smi00] M. D. Smith. "Overcoming the Challenges to Feedback-Directed Optimization," *Proc. ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, January 2000.

[Spe00] SPEC CPU2000 benchmark, December 2000, *Available from http://www.spec.org/osg/cpu2000/*.

[Spr02] B. Sprunt. "Pentium4 Performance Monitoring Features," *IEEE MICRO, Vol. 22, No. 4*, July-August 2002.

[ZWG+97] X. Zhang, Z. Wang, N. Gloy, J.B. Chen, and M.D. Smith. "System Support for Automatic Profiling and Optimization," *Proc. 16th ACM Symposium on Operating Systems Principles*, October 1997.

Optimal Global Scheduling for Itanium™ Processor Family

Sebastian Winkel
Saarland University
Saarbrücken, Germany
sewi@cs.uni-sb.de

Abstract

Global scheduling with integrated decisions about speculation and predication for Itanium™ Processor Family (IPF) is widely known as a complex and challenging task.

Compilers find it especially difficult to use the proper amount of speculation and code motion, as both techniques increase the demand for execution resources. If applied too conservatively, free execution slots are wasted, contrary to the EPIC philosophy. If applied too aggressively, resource shortage can spoil the benefit. It is unknown how well state-of-the-art scheduling heuristics perform here.

We take a very precise approach to this problem: we reformulate it as a combinatorial optimization problem and apply integer linear programming (ILP) to obtain provably optimal and correct solutions. We integrate code motion with automated generation of compensation code and control speculation into the ILP model.

Since the performance of IPF is highly compiler-dependent, optimal schedules promise a speedup for compute-intensive applications, as well as some theoretically funded insights into the potential of the architecture.

Early experiments with several functions from the SPEC benchmarks show substantial improvements: Our post-pass optimizer reduces the schedule lengths produced by Intel's compiler by 20-30%.

1 Introduction

One of the major challenges of EPIC code generation is to find a proper balance between (speculative) code motion and resource demand. Code motion is applied to decrease the schedule length, but it can also increase the resource demand in several manners:

First, as Fig. 1 shows, a speculative upward movement of an instruction like from block B to A (I) has the effect that this instruction occupies an execution slot unnecessarily on the path A-C-D.

Second, an upward movement across a join like from D to B (IV) enforces the placement of a *compensation copy* of the instruction in block C (KIV). The resource demand does not increase for a single path, but for the total schedule. Moreover, control speculative loads require additional check instructions.

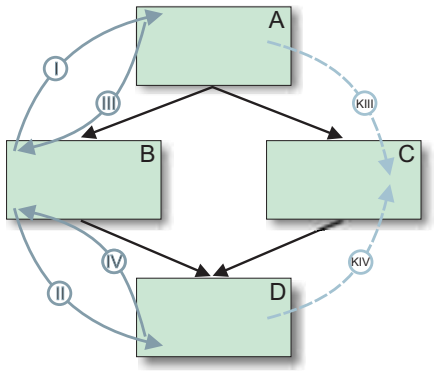


Figure 1: Code motion: upward (I+IV), downward (III+II), speculative (I+II), non-speculative (III+IV).

One might think that, with the plenty of execution units available in EPIC processors, resource shortage in general is not an issue. However, there is a two-fold effect how resource pressure increases as these transformations are applied:

Not only does the demand for execution slots increase as described, but also the supply of execution slots decreases with the schedule length. Fig. 2 gives an idealized illustration of this effect, with the optimal schedule located at the intersection point of the supply and demand curves.

While a "hesitant" scheduler could apply code motion and speculation too conservatively and waste opportunities, an overeager code generator could decide too aggressively and thereby exceed the available resources (point A). This could spoil the benefit of these techniques as the final schedule is formed (to point B).

Besides from these difficult trade-offs, even simple local instruction scheduling is an NP-complete problem where heuristics only deliver approximations.

We use integer linear programming to obtain *globally optimal* and *provably correct* solutions to this problem.

"Optimal" means schedules with minimal length here, as defined precisely later in Sec. 4.2. It is not guaranteed that

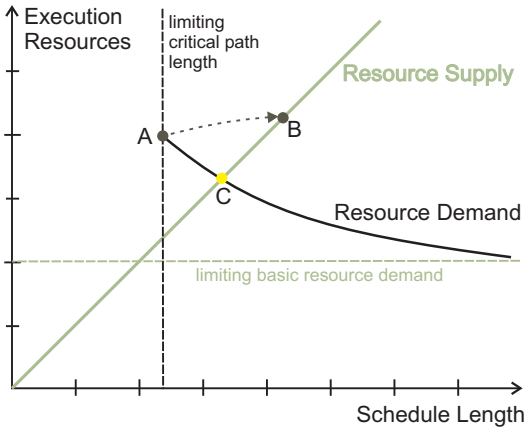


Figure 2: Trade-off between schedule length reduction and resource demand.

these minimal schedules necessarily deliver maximal performance in practice – there are many other factors influencing performance – however, on a statically scheduled architecture, there should be a strong correlation between schedule length and performance.

Optimal solutions need their time – up to a few minutes –, and so we do not intend to replace the scheduling heuristics of general purpose compilers. We have two different objectives:

- We see an application as an optimization tool for performance-critical software components like encryption routines, probably as a postpass solution.
- We expect answers to some very basic questions that are still partly open for EPIC architectures:
  How much parallelism can be statically extracted by using code motion, predication and speculation?
  How well perform scheduling heuristics like wavefront scheduling [BM00]?

The rest of the paper is organized as follows: Section 2 gives a short introduction to integer programming. The basics of the ILP model and extensions are described in Sections 4 and 5, respectively. It is assumed here that the reader is familiar with fundamentals of IPF like static scheduling and control speculation. Section 6 presents the experimental results and Section 7 concludes the paper.

2 Integer Linear Programming

Since the invention of the simplex algorithm by George B. Dantzig over fifty years ago [Dan51], *linear programming* has developed to an indispensable tool for the formulation and solution of optimization problems.

This applies especially to the unequally more powerful – and unequally more difficult to solve – *integer* linear programming (ILP), whose potential was almost immediately recognized after its discovery in the fifties [BFG+00]. But insufficient hardware and software have soon led to some disillusionment and to the perception that ILP has very limited practical applicability.

In the last years, however, this situation has changed "dramatically" due to advances in solution algorithms and ILP formulations [BFG+00]. This is also confirmed by our own experiences.

Integer linear programming (ILP) minimizes a linear objective function subject to a system of linear constraints given by $P_R = \{x | Ax \leq b, x \in \mathbb{R}^n_+\}$ with $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$:

$$\begin{array}{lrcl} \min & z_{IP} & = & c^T x \\ & x & \in & P_R \cap \mathbb{Z}^n \end{array} \quad (1)$$

The integer points $P_I = P_R \cap \mathbb{Z}^n$ form the *feasible solutions* or the *search space*. Computing an optimal solution is NP-complete, but the *relaxed problem* without the integrality restriction of equation (1) can be solved in *polynomial time* [NW88].

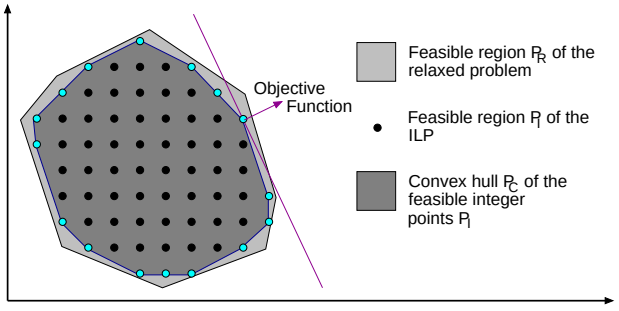


Figure 3: Feasible regions.

Note that if the polyhedron P_R would be made equal to the convex hull of the feasible integer points, then also the integer problem could be solved in polynomial time (see Fig. 3). Though equality usually cannot be achieved in practice, it is important for the solution efficiency to find a *tight* ILP formulation where P_R is close to this convex hull.

3 Related Work

Besides from many heuristics like [BM00, BR91], there exist only few ILP-based exact approaches to instruction scheduling:

Wilson et al. [WGB94] and Chang et al. [CCK97] simultaneously perform scheduling and register allocation; the former also include code selection. Both works show experimental results only for very small examples, with solution times of several seconds.

Wilken et al. [HLW00] show that by using a tight ILP formulation and clever precomputations, solution times of

less than 0.1 seconds can be achieved for scheduling basic blocks of a hundred and more instructions. Kästner and Winkel [KW01] combine scheduling and bundling for the Itanium architecture in a two-phase approach.

All works mentioned only deal with local scheduling – the only ILP model known to us that tackles acyclic global scheduling is used by the postpass optimizer PROPAN for DSPs [Käs00]. However, it allows no disjoint control flow paths in the ILP and code motion only between control equivalent basic blocks.

A local scheduler based on optimal approaches without ILP is presented by Haga and Barua [HB01]. In contrast to most earlier work, they integrate template selection into the scheduling process to minimize the number of NOPs on EPIC architectures.

4 The Basic ILP Model

Our goal is to formulate an ILP model for the given input program where every integer point inside the polyhedron corresponds to a possible schedule and vice versa. We say that a schedule is *feasible* if the corresponding point is a feasible solution of the ILP model.

We first collect some basic requirements for the ILP formulation. It should be kept in mind that, for search-based methods like ILP, solution efficiency is an important issue, and the last three points in the list take this into account.

Remark 1 The ILP model should be:

1. **correct:** no incorrect schedule is feasible
2. **complete:** at least one optimal schedule must be feasible
3. **compact:** as many non-optimal schedules as possible are excluded from the search space
4. **simple:** as much abstraction and unification should be used to have as few variables and constraints as possible
5. **efficient:** the inequalities should describe a tight polyhedron (see. Sec. 2) □

During the following introduction of the basic structure of the model, the emphasis lies on the correctness.

4.1 Correctness

At first we assume that the scheduling region is acyclic; we will expand on loops later in section 5.2. Let $G_B = (\mathcal{B}, E_C, \mathcal{B}_{entry}, \mathcal{B}_{exits})$ be the basic block graph of the scheduling region with the set of basic blocks $\mathcal{B}$ and the control flow edges E_C. Entry and exit blocks are given by $\mathcal{B}_{entry}$ and $\mathcal{B}_{exits}$, respectively. We call block D a (direct) successor of C if there is a path from C to D in G_B (consisting of one edge); the definition of predecessor is analogical.

Global data dependences are given by the acyclic data dependence graph $G_D = (V, E_D)$. Each edge $e \in E_D$ has a latency lat_e associated with it, where false and memory dependences[1] have the latency zero. On the Itanium architecture, execution units generally have a throughput of one instruction per cycle.

We can view global scheduling as a *transformation* between global schedules which rearranges instructions, but does not change the control flow structure (although it may empty some blocks). Hence the set of *program paths* – paths which go from an entry block to an exit block through the scheduling region – remains unchanged.

This allows us to take a path-based view of correctness and say that a transformation from schedule δ to δ' is correct if the same computations (and probably exceptions) are performed in both schedules along every program path.

To be more precise, this is the case when all instructions that occur along a path in δ also occur there in δ', and when all dependences between these instructions are preserved. Additionally, *non-speculative* instructions may only appear on a path in δ' if they appear there in δ, too.

For each instruction $n \in V$, we call the block where it originates from before scheduling *source block*, denoted by $s(n)$. Code motion moves the instruction from this source block to a *destination block*. Possible destination blocks are all predecessors and successors of the source block in G_B. We denote $\Theta_{spec}(n)$ as the set of those *speculative destination block candidates*[2] for instruction n. The range of destination blocks is further limited for non-speculative (and unpredicated) instructions like the following:

- unsafe loads [BRS92],
- stores,
- concurrent definitions,
  where a value can be defined by more than one instruction, depending on control flow. This is discussed in Section 5.1.
- branches, which are considered as special instructions and not included in the set V.

For those instructions a speculative placement can be ruled out if the source block dominates and postdominates the destination block for downward and upward code motion,

[1] i. e. dependences resulting from accesses to memory locations, for example between a store and a load.

[2] In the following, we often call destination block candidates simply "destination blocks".

respectively. Accordingly, we define a set $\Theta(n)$ of (actual) *destination block candidates* which is the same as $\Theta_{spec}(n)$ except that the following blocks are excluded for non-speculative instructions:

- all predecessors of $s(n)$ which are not postdominated by $s(n)$ and
- all successors of $s(n)$ which are not dominated by $s(n)$

Each instruction can be scheduled into parallely executable *instruction groups* in its destination blocks. Within each destination block A, there is a range $G(A) = \{1, \ldots, \mathbb{G}_A\}$ of possible successive groups (or *time steps* / *cycles)* given. Our ILP model uses the following main decision variables to model this:

$$x_n^{At} = 1 \quad \Leftrightarrow \quad \text{A copy of instruction } n \text{ is scheduled at time step } t \text{ in } A$$

These binary variables are generated for all instructions n, all destination blocks $A \in \Theta(n)$ and all time steps therein.

In a correct schedule, every path through the source block of an instruction must contain a copy of the instruction. To express this later in an equation, we employ binary variables for all $n \in V$ and all $A \in \Theta_{spec}(n)$ with the following semantics:

$$a_n^{\uparrow A} = 1 \quad \Leftrightarrow \quad \text{A copy of instruction } n \text{ is scheduled on all program paths through } s(n) \text{ before } A$$

We need to couple the x and the a variables with constraints to model the described semantics. This is done inductively using the following observation:

Let $B \in \Theta_{spec}(n)$ be a destination block and $A \in \Theta(n)$ be a direct predecessor of B. If n is scheduled on all paths through $s(n)$ before B, then it is *either* scheduled at A *or* on all paths through $s(n)$ before A. This is expressed by the following equations which are added to the model for all instructions n, all blocks $B \in \Theta_{spec}(n)$ and all of B's direct predecessors A in $\Theta(n)$:

$$a_n^{\uparrow B} = a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} \tag{2}$$

In the case that a predecessor A is only a speculative destination block and not element of $\Theta(n)$, we generate the equation without the sum. If B has no predecessors at all, we set $a_n^{\uparrow B} = 0$.

It should be clear that equation (2) realizes the desired semantics. These two classes of variables are now sufficient to model global scheduling:

First, we have to ensure that every program path through the source block of an instruction contains a copy of it. This is done by the following *assignment constraints*, where Ω is a new, empty pseudo block which is added as a successor of all exit blocks:

$$a_n^{\uparrow \Omega} = 1 \qquad \forall n \in V \tag{3}$$

Further, if an instruction n is dependent on m, it must appear after m on every path. *Globally*, this can be achieved by adding the following *precedence constraints* for all $(m, n) \in E_D$ and for all $A \in \Theta_{spec}(m) \cap \Theta_{spec}(n)$:

$$a_n^{\uparrow A} \leq a_m^{\uparrow A} \tag{4}$$

The correctness proof below will demonstrate the functioning of these inequalities. To ensure that dependences *inside* a basic block are met, we adapt the proven and efficient (tight) *local precedence constraints* from [GE93, KW01, Win01]:

$$\sum_{\substack{t_n \leq t \\ t_n \in G(A)}} x_n^{At_n} + \sum_{\substack{t_m \geq t - lat_e + 1 \\ t_m \in G(A)}} x_m^{At_m} \leq 1 \tag{5}$$

$$\forall e = (m, n) \in E_D,$$

$$\forall t \in \{t' + lat_e - 1 \,|\, t' \in G(A)\} \cap G(A)$$

Fig. 4 (left) helps to understand the intuition behind these constraints with a simple example consisting of two dependent instructions m and n and three time steps. The bordered area represents the variables on the left-hand side of inequality (5) (for one instance with $t = 2$) – if an x_m and an x_n variable in this sum were one, this would imply that m is scheduled at time step two or three and n at one or two, which would violate the dependence. This violation is excluded by setting the sum less than or equal to one. In [Käs00] it has been shown that any infeasible instruction ordering is excluded but no feasible solution discarded.

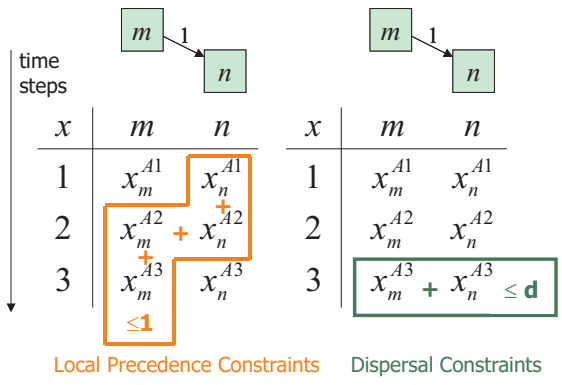


Figure 4: Simple example for the local precedence and dispersal constraints.

Further constraints must ensure that the number of instructions scheduled at one time step does not exceed the target processor's execution resources. On Itanium processors, this number is generally limited by the dispersal window size d (six for Itanium 2). With help of the inverse Θ^{-1} of Θ, we can formulate that not more than d instructions may be issued in one cycle:

$$\sum_{n \in \Theta^{-1}(A)} x_n^{At} \leq d \qquad (6)$$
$$\forall A \in \mathcal{B},\ \forall t \in G(A)$$

Additionally, the number of instructions for a specific execution unit type is limited. The Itanium 2 has ten functional units: Two load units (M0+M1), two store units (M2+M3), two integer units (I0+I1), two floating-point units (F0+F1) and three branch units. Different instruction types can be executed on different subsets of execution units, for example A(LU)-type instructions on all six memory and integer units.

For complexity reasons, we do not integrate resource binding into the ILP, i. e. the ILP solver does not decide whether, for instance, an A-type instruction is executed on a memory or integer unit. In contrast to our earlier work [KW01], this decision is now left open to a later bundling phase. We can afford to do so since the Itanium 2 has significantly less restrictions than the first generation (e. g. full ALU bypassing, much more flexible bundling [NH02, Int02]).

Instead, constraints ensure that a later binding of instructions to execution units is *possible*. For this we define a set $\{\mathcal{M}0,\mathcal{M}1,\mathcal{M}L,\mathcal{M}2,\mathcal{M}3,\mathcal{M}S,\mathcal{M},\mathcal{I}0,\mathcal{I}1,\mathcal{I},\mathcal{A},\mathcal{F}0,\mathcal{F}1,\mathcal{F}\}$ of *abstract execution units*, which themselves are sets of execution units; e. g. $\mathcal{M}L = \{\mathsf{M0}, \mathsf{M1}\}$ and $\mathcal{I}0 = \{\mathsf{I0}\}$. These sets are hierarchically nested according to Fig. 5, i. e. each set forms the union of its successors. We denote $p(\mathcal{U})$ as the predecessor of $\mathcal{U}$ in the hierarchy.

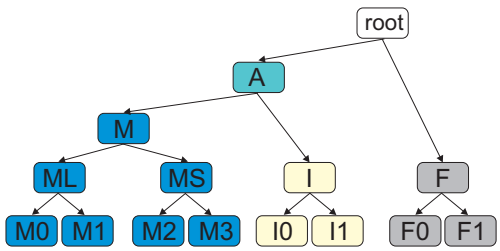


Figure 5: Hierarchy of abstract execution units.

For each abstract execution unit $\mathcal{U}$, let the set $\bigsqcup(\mathcal{U}) \subseteq V$ contain those instructions which can be executed on *at least one* execution unit in $\mathcal{U}$, and $\bigsqcap(\mathcal{U})$ those instructions which can be executed on *all* execution units in $\mathcal{U}$. Then the following inequalities form, as shown in [Win01], valid *resource constraints* when created for all abstract units $\mathcal{U}$, all blocks A and all time steps t therein:

$$\sum_{\substack{n \in \bigsqcup(\mathcal{U}) \setminus \bigsqcap(p(\mathcal{U})) \\ \cap \Theta^{-1}(A)}} x_n^{At} \leq |\mathcal{U}| \qquad (7)$$

A small example shows that the idea behind these constraints is straightforward. Consider three loads l_1, l_2, l_3, four A-type instructions a_1, a_2, a_3, a_4 and two instructions i_1 and i_2 which can only be executed on I0. Then the following constraints are generated for time step 1:

$$\begin{aligned} x_{l_1}^{A1} + x_{l_2}^{A1} + x_{l_3}^{A1} + x_{a_1}^{A1} + & \\ x_{a_2}^{A1} + x_{a_3}^{A1} + x_{a_4}^{A1} + x_{i_1}^{A1} + x_{i_2}^{A1} &\leq 6 \quad (\mathcal{A}) \\ x_{l_1}^{A1} + x_{l_2}^{A1} + x_{l_3}^{A1} &\leq 2 \quad (\mathcal{M}L) \\ x_{i_1}^{A1} + x_{i_2}^{A1} &\leq 1 \quad (\mathcal{I}0) \end{aligned}$$

The polytope for global scheduling is now complete. Regarding the demand for simplicity from Rem. (1), it should be noted that we need only two classes of variables and six sorts of constraints for the basic model. Under the assumption that $|V| \leq |E_D|$ and with $\mathbb{G} = \sum_{A \in \mathcal{B}} \mathbb{G}_A$, we need $\mathcal{O}(\mathbb{G} \cdot |V|)$ variables and $\mathcal{O}(\mathbb{G} \cdot |E_D|)$ constraints.

We conclude with a correctness proof:

Corollary 1 *For any feasible schedule holds: Instruction n is scheduled at block A if and only if the variables $a_n^{\uparrow}$ have value one for all successors of A in $\Theta_{spec}(n)$ and value zero for A and its predecessors in $\Theta_{spec}(n)$.* □

PROOF Follows inductively from equation (2). ■

Theorem 1 (Correctness) *Every integer point that satisfies constraints (2)-(7) corresponds to a correct global schedule.* □

PROOF Let the corresponding schedule and a program path P be given. It is sufficient to show for any instruction n with source block on P that it (1) appears along this path in the schedule, and that (2) dependences on other instructions are not violated.

The first claim follows directly from constraint (3) since P passes through Ω. So let n be scheduled in block A on P.

For the second claim, we show that for each dependence $(m, n) \in E_D$ (with $s(m) \in P$), a copy of m is scheduled *before or in* A on P and no copy *after* A on P.

We first note that all blocks on P are in $\Theta_{spec}(m)$ and $\Theta_{spec}(n)$. From the corollary we know that the variables $a_n^{\uparrow}$ have value one for all successors of A on P and value zero for A and its predecessors on P. With (4) it follows that the variables $a_m^{\uparrow}$ are one for all successors of A on P, too.

Consequently, no copy of m is scheduled in a successor of A on P (again with the corollary) and, since $a_m^{\uparrow}$ is one for the *direct* successor of A on P, m is scheduled *before or in* A on P .

Concerning dependences inside a basic block and the resource limitations (constraints (5) and (7), respectively), references to the correctness proofs have been given in the text above. ■

The correctness of the generated code follows directly from this correctness proof. In general, any schedule is correct if it is a feasible solution of the ILP (which can be checked in time that is linear in the size of the ILP).

This property can also be used to validate schedules produced by heuristics. It is an inherent advantage of this approach which builds not on an algorithm, but on a precise mathematical model.

4.2 Completeness

At least one optimal and correct schedule must be feasible in order to be found by the ILP solver. We can show that with the given ILP model – independent of how optimality is defined – *any* correct schedule is feasible:

Theorem 2 (Completeness) *Let a correct schedule be given where no instruction is placed twice on any path. Then the corresponding integer point is a feasible solution of the ILP model.* □

We must leave out the proof here for lack of space. It is important for the proof that we exclude the possibility to schedule an instruction twice on a path in the theorem. This restriction can be lifted for *P-ready code motion* later (see Section 5.3).

There are several other aspects that could restrict completeness, but are out of the scope of the above theorem.

One concerns the number of time steps $\mathbb{G}_A$ given for each basic block A. The ILP solver could choose to grow less frequently executed blocks by moving code into them – this possibility should not be limited by a too small $\mathbb{G}_A$. This is a sensitive issue since this value heavily affects the size and thereby the solution times of the produced ILPs. A save choice is to collect all instructions which could possibly be moved into the block, $\Theta^{-1}(A)$, and compute via list scheduling an upper bound on the length of an optimal local schedule of all these instructions.

Another point concerns the destination blocks of non-speculative instructions: the latter can be executed speculatively if they are guarded by predicates which eliminate the speculativeness; this allows an extension of the range of destination blocks.

For *upward motion* of an instruction, such a predicate register can be found as follows: For all control flow edges $(A, B) \in E_C$ where B dominates the source block of the instruction and A not, the qualifying predicate of the branch associated with the edge is a candidate. Guarded by this predicate register, the instruction can be safely moved to A and all of its predecessors (but there is a new data dependence on the compare which generates the predicate value).

We perform a similar extension for *downward motion* of an instruction: let the set $\mathcal{E}$ contain the source block and all control equivalent basic blocks, and let $A \in \mathcal{E}$ be the "top" block of them, i. e. the one that has no predecessors in A. We examine the control flow edges leading to A – if it is only one edge with a predicate associated with it, then the instruction is executed if and only if this predicate is true. When used as a qualifying predicate of the instruction, the latter can be moved downwards arbitrarily far.

This way we determine for each new destination block a predicate register which must be used as qualifying predicate if the instruction is scheduled there; so we include predication in our model as a *side-effect of code motion*.

We finally define the notion of optimality exactly: Optimization goal is to minimize the *global schedule length*, which we define as the sum of the schedule lengths of all basic blocks, each weighted by the execution frequency of the block (which we assume to be given).

To integrate this into the model, we introduce a new integer (but not binary) variable T_A which is greater than or equal to the length of basic block A in the schedule:

$$\sum_{t \in G(A)} t \cdot x_n^{At} \leq T_A \qquad \forall n \in V,\, \forall A \in \Theta(n)$$

With the execution frequency of block A given as f_A, the objective function can be written as:

$$\min_{A \in \mathcal{B}} \sum f_A \cdot T_A \tag{8}$$

We do not integrate register allocation into the model since IPF offers a large amount of 128 architecturally visible registers. Hence the interaction between register allocation and scheduling is relatively low so that it can be done in separate phases efficiently.

5 Extensions

5.1 Speculation

Non-speculative instructions are limited in their scope of code motion as they may not be executed unnecessarily. In general, there are two reasons why the unnecessary execution of an instruction could harm correctness: First, it could trigger a false exception, which concerns mostly memory instructions. Second, it could overwrite a live value; this applies to stores and to *concurrent definitions* [Käs00]:

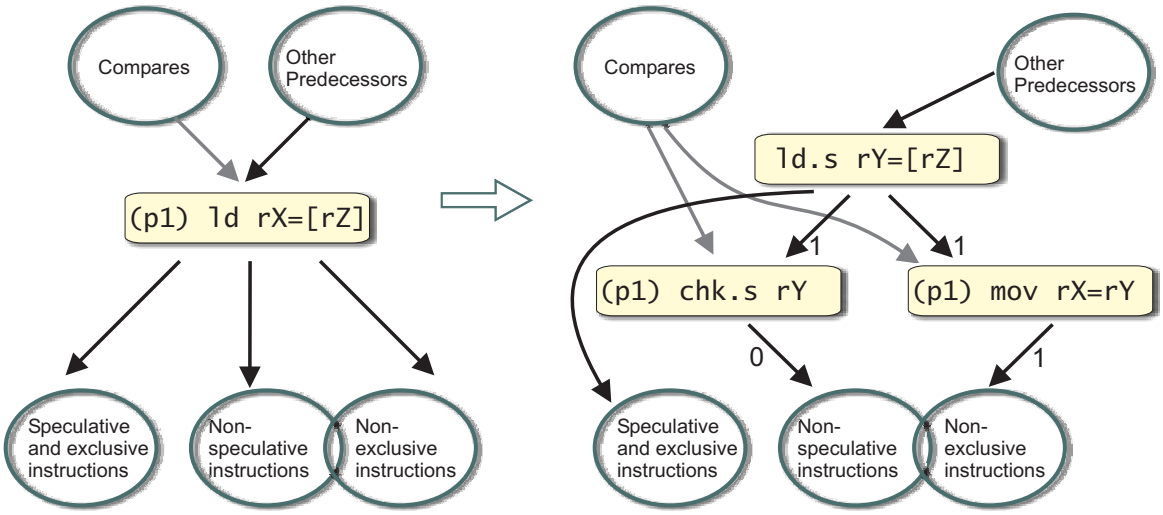


Figure 6: Control speculation with register renaming before (left) and after application (right).

Definition 1 (Concurrent Definitions) Two instructions are called *concurrent definitions* if they write the same register and if there is a use of that register reachable by both definitions (i. e., both are in the use-def chain of the use).

A use of the register is called *exclusive* if it is reachable only by one definition. □

Fig. 6 presents a scheme that allows to execute a load which is also a concurrent definition speculatively – at the price of two additional instructions plus recovery code. The left-hand side shows the original load where the arrows represent sets of true data dependence edges. We distinguish three groups of uses of the register `rX` written by the load: speculative[3] uses which are also exclusive with respect to `rX`, non-speculative instructions and non-exclusive uses, where the latter two groups may overlap, as depicted in the figure.

The right-hand side shows how the load is replaced by a control-speculative version `ld.s` which writes to a new temporary register. All speculative and exclusive uses can directly read the temporary register and thereby be speculated with the load.

The check instruction `chk.s` detects an exception deferred by the speculative load `ld.s` and must be scheduled before all non-speculative uses. The new `mov` instruction moves the loaded value from the temporary to the original register. It must analogically be executed before all non-exclusive uses.

These two new instructions are non-speculative and hence must be guarded by the predicate of the former non-speculative load.

The scheme is comprehensive in the sense that it is also possible to speculate a concurrent definition that is not a load and vice versa – the `chk` and the `mov` are then dropped, respectively. It also allows to cascade several dependent speculative instructions – an example of this is discussed later in Sec. 6.

We integrate the possibility to use this kind of speculation into the search space so that the ILP solver can optimally decide whether to employ it. It decides between two mutually exclusive instruction groups: The first consists of the normal load and the second of the speculative version and the `chk` and/or the `mov`. Either the first or the second group must appear in the final schedule, and their dependences must be obeyed.

To realize this, we include the instructions of *both groups* in the ILP and define a binary variable $usespec$ as a "speculation switch". The right-hand side of the assignment constraints (3) is replaced by $(1 - usespec)$ and $usespec$ for instructions from the first and second group, respectively.

To switch the global precedence constraints on and off, we add the first and the second term to the right-hand side of (4) for all dependences involving instructions from the first and second group, respectively. The terms *relax* these inequalities if the dependences are "switched off". All other constraints of the ILP are not affected by these mutually exclusive instruction groups.

It should be noted that the branch to recovery code is not taken if the speculation fails, but only if the load triggers an exception. These cases are very rare and expensive anyway, so the cost of recovering does not need to be taken into account. This is different for data speculation, which is implemented very similarly, but we cannot go into details here for lack of space.

[3] Analogous to the term "non-speculative", we call instructions "speculative" if they can be executed speculatively.

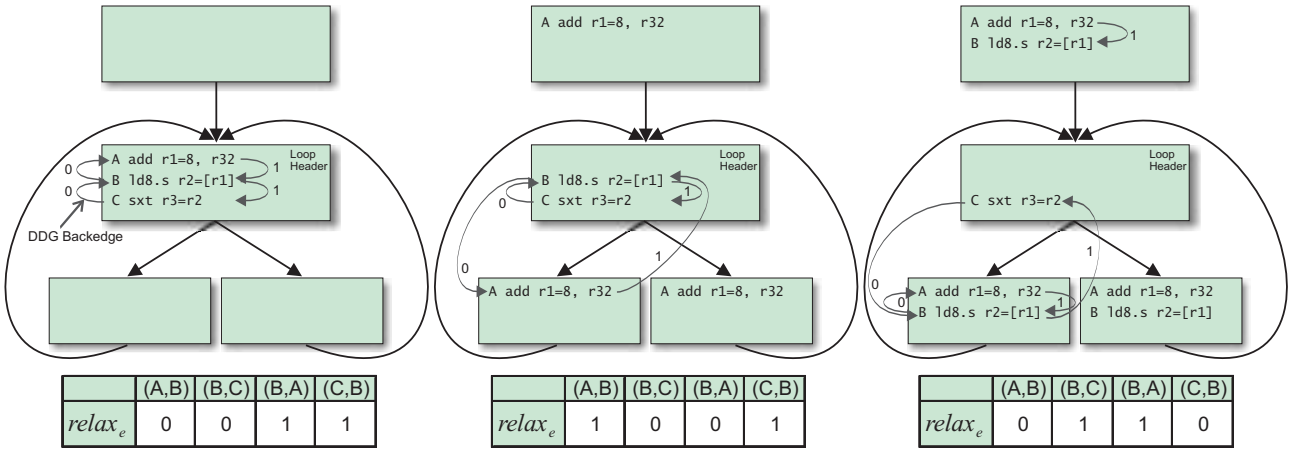


Figure 7: Example for cyclic code motion. The DDG backedges with latency zero are WAR dependences. The relax terms in the table show how these and the other dependences are switched on and off as the `add` and the `load` are cyclically moved ($relax_e = 0$ and $relax_e = 1$, respectively) .

5.2 Cyclic Code Motion

Unlike many other global scheduling algorithms, we do not limit the scheduling scope to acyclic regions. Loops are such an essential element of every program that this restriction would limit the search space and thereby our objective of truly optimal schedules too strongly.

Consequently, we allow code motion *into* loops and *out of* loops. The former can only be allowed if the instruction

- does not write a value which is live at the loop header
- is *multiple executable* without a changing semantics, which is not the case for instructions where results and operands overlap, like `add r1=1,r1`.

In most cases, we cannot move an instruction completely out of a loop[4]. For instance, upward code motion of an instruction requires that copies of it are moved not only upwards into blocks above the loop header, but also *along every backedge* to the bottom blocks of the loop and their predecessors. We call this kind of code motion *cyclic*.

Fig. 7 shows a small example where the `add` (middle) and subsequently the `load` (right) are moved cyclically.

One benefit of cyclic motion is that parts of the first iteration of the loop can be overlapped with code preceding the loop and thereby be executed earlier. If the loop-carried dependences permit it, the cyclically moved code can also be overlapped inside the loop body, which may lead to a drastically reduced critical path here.

Cyclic code motion can then be regarded as a simple variant of software pipelining without fill overhead. In practice, there are still many loops where software pipelining can or should not be applied – for example if they contain other loops or function calls, or if they have low trip counts – cyclic code motion can then help alleviate the inefficiencies static scheduling suffers from in these cases.

We currently have integrated only cyclic upward code motion for speculative instructions into the ILP. Little changes are necessary and no new variables and constraints need to be introduced. For an instruction m originating from a loop with header H, the variable $a_m^{\uparrow H}$ is assigned a special meaning: m is cyclically moved if and only if $a_m^{\uparrow H} = 1$. This choice is left open to the ILP solver.

As cyclic code motion changes the order of instructions inside the loop body, the data dependences change, too. This is modeled by adding a relax term to the right-hand side of the inequalities (4) and (5), which is $relax_e := 1 + a_m^{\uparrow H} - a_n^{\uparrow H}$ for DDG backedges (m, n) and $relax_e := a_m^{\uparrow H} - a_n^{\uparrow H}$ for normal DDG edges. How the dependences are switched on and off is illustrated exemplarily in Fig. 7.

5.3 Further Work

We briefly sketch several extensions we are currently developing and testing:

Partially-ready code motion can be supported, as described in [BM00]. Global data dependences can vary with control flow, and P-ready code motion allows those to be ignored which are not relevant on a program path. This can lead to a further schedule length reduction.

Long-latency instructions like floating-point commands are currently not properly supported as the global effect of these latencies between basic blocks is not allowed for. An extension is being developed.

Software pipelining [AJLA95]: A model to minimize the length of the software pipeline and the kernel has been presented in [Win01].

[4] This would only be possible for loop invariant instructions, but we expect that this optimization has already been done.

Stall minimization is a code reordering technique which is applied in a separate phase after scheduling and expands the distances between loads and their nearest use, while preserving the optimality of the schedule. The objective function of the ILP can be modified for this purpose. The reordering minimizes the stall cycles due to cache misses, which often account for a large proportion of the whole execution time on statically scheduled architectures.

6 Experimental Results

We have implemented all described modelings and conducted several experiments. With the help of Intel's VTune Analyzer, we first located some hot routines in the SPEC CINT2000 benchmark. Our selection is currently limited to four routines since floating-point instructions, function calls and software pipelining are not yet supported by the implementation.

The selected routines were compiled to assembly with Intel's compiler 6.0.1 for Itanium. We used full optimization (`-Ox`) and the switch `-G2` to obtain code for Itanium 2.

The assembly files are directly input to our optimizer. The latter reconstructs control flow, data dependences and also reads the execution frequency estimates for function (8) which are delivered by Intel's compiler although no profile feedback could be used. It also undoes all usages of control speculation and performs register renaming to remove all false dependences which would otherwise restrict code motion. We also perform several automated optimizations to make the search space *compact*, e. g. we exclude possibilities for code motion which cannot be utilized in any correct and optimal schedule.

The advantage of the postpass approach is that we can compare the results directly with those produced by Intel's state-of-the-art compiler [DKK+99]. Since we lack an Itanium 2 machine, however, this comparison currently can only occur statically.

A drawback of the postpass approach is that no information about memory disambiguation is available. Hence memory dependencies must be reconstructed conservatively. However, in *longest_match* and *add_penal* one and six instances of data speculation, respectively, are applied in the optimal schedule to overcome such dependences. From the source code it is evident that no address aliasing is possible in these cases, so the cost of failed data speculation does not need to be taken into account. Nevertheless, more comprehensive memory disambiguation information is indispensable for further experiments [GLS01].

Table 1 shows for each routine the number of basic blocks, loops and the number of instructions and speculation usages at different stages: as produced by Intel's compiler ("Int."), as possibilities included in the ILP ("ILP"), and finally as occurring in the optimal schedule ("Opt.").

The number of instructions included in the ILP, #Ins.ILP, is generally higher than the original number, #Ins.Int, because additional instructions like speculative loads with their checks are generated. Since not all of those speculation possibilities are used in the optimal schedules (#Spec.Opt. < #Spec.ILP), #Ins.Opt is usually lower than #Ins.ILP.

An exception is *get_bb_from_scratch*, where multiple compensation copies of instructions (see Sec. 1) lead to a higher count in the final schedule.

Table 2 shows the size of the ILPs after *presolve*, a preprocessing by the ILP solver which removes redundant constraints and variables. The solution times range between 6 and 20 seconds with the ILP solver CPLEX 8.0 [cpl02] on a 333-MHz-UltraSparc II.

Fig. 8 shows the results – the average improvement is about 30%. Further improvements – although diminishing – occur when we define a target processor with more execution units. This indicates that the schedules for Itanium 2 are still resource-bound.

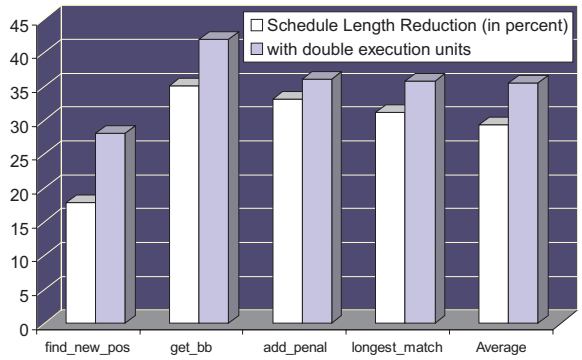


Figure 8: Global schedule length improvements over Intel's compiler on the same Itanium 2 target processor (left bars) and on a processor with the double number of execution units (right bars).

Fig. 9 shows exemplarily for the entry block of *get_bb_from_scratch* how such improvements are possible. The reduction of the critical path is achieved by speculating two concurrent definitions

- `(p7) add r10=16,r17`
- `(p8) shladd r10=r23,3,r16`

and three loads

- `(p7) ld8 r14=[r10]`
- `(p8) ld8 r14=[r10]` (both concurrent)
- `(p8) ld8 r16=[r10].`

Routine Name (Benchm.)	#Ins.Int.	#Ins.ILP	#Ins.Opt.	#Spec.Int.	#Spec.ILP	#Spec.Opt.	#BB/Loops
get_bb_from_scratch (vpr)	122	137	142	0	8	7	3/1
add_penal (twolf)	109	163	123	0	26	11	9/1
find_new_pos (twolf)	107	114	108	0	8	1	11/0
longest_match (gzip)	154	202	171	12	37	18	20/2

Table 1: Input routines.

Routine Name (Benchm.)	#Constraints	#Variables	Solution Time (in seconds)
get_bb_from_scratch (vpr)	1981	1465	6
add_penal (twolf)	2666	1603	13
find_new_pos (twolf)	1457	873	18
longest_match (gzip)	2878	1794	20

Table 2: Numbers of constraints, variables and the solution times of the ILPs.

Cycle	Input Schedule	Output Schedule

1	`add r11=@gprel(net#),gp`
	`add r9=@gprel(duplicate_pins#),gp`
	`add r10=@gprel(unique_pin_list#),gp`
	`sxt4 r23=r32`
		`add r11=@gprel(net#),gp`
		`add r9=@gprel(duplicate_pins#),gp`
		`add r10=@gprel(unique_pin_list#),gp`
		`sxt4 r23=r32`

2	`ld8 r8a=[r9]`
	`ld8 r3a=[r11]`
	`shl r22=r23,5`
		`ld8 r8a=[r9]`
		`ld8 r3a=[r11]`
		`shl r22=r23,5`

3	`shladd r19=r23,2,r8a`
	`add r17=r3a,r22`
		`shladd r19=r23,2,r8a`
		`add r17=r3a,r22`
		`ld8.s r16=[r10]`

4	`ld4 r18=[r19]`	`ld4 r18=[r19]` **`add r10b=16,r17`** **`shladd r10a=r23,3,r16`**
5	`cmp4.ne.unc p8,p7=r18,r0`	`cmp4.ne.unc p8,p7=r18,r0` **`ld8.s r14b=[r10b]`** **`ld8.s r14a=[r10a]`**
6	`(p7) add r10=16,r17` `(p8) ld8 r16=[r10]`	**`(p8) chk.s r16`** **`(p7) mov r14=r14b`** **`(p8) mov r14=r14a`** **`(p7) mov r10=r10b`** **`(p8) mov r10=r10a`**

7	`(p7) ld8 r14=[r10]`
	`(p8) shladd r10=r23,3,r16`
		`(p7) chk.s r14b`
		`(p8) chk.s r14a`
		`ld4 r26=[r14]`
		`add r32=4,r14`

8	`(p8) ld8 r14=[r10]`	
9	`ld4 r26=[r14]` `add r32=4,r14`	

Figure 9: Example for schedule length reduction through control speculation. Shown is a slice from *get_bb_from_scratch* before and after optimization. New instructions are written in bold; new, renamed registers have an appended '`a`' or '`b`'.

Cyclic code motion accounts for another large part of the improvements: when switched off, the numbers from Fig. 8 drop to 17% on the average. In *longest_match*, for example, cyclic code motion is applied to six instructions from the outer loop and to another six from the inner loop, which significantly reduces the critical path lengths in both loops.

Finally, Fig. 10 shows how the instruction-per-clock rate moves closer to the maximum of six as the schedule length decreases and more slots are used for speculation.

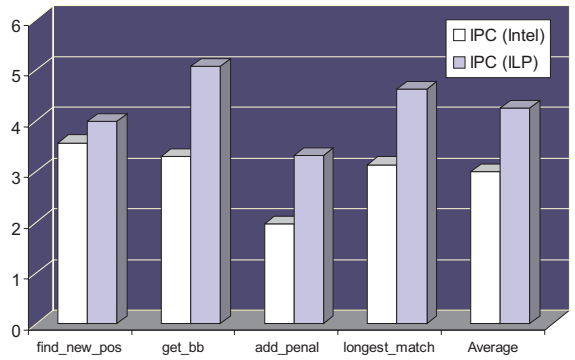


Figure 10: The optimal schedules (right bars) come closer to the theoretical maximum of six instructions per cycle.

7 Conclusion and Outlook

To our knowledge, we are the first to present a comprehensive formal description of global scheduling with integrated generation of compensation code, cyclic code motion and speculation.

Much effort has been spent to make the ILP model not only correct and mostly complete, but also simple and efficient. As a result, we are able to compute optimal schedules for regions with 100 to 200 instructions within a few seconds.

Our early experimental results are very promising: the ILP solver could, applied as a postpass optimizer, reduce the schedule lengths of several functions about 20-30% compared with Intel's sophisticated compiler. Cyclic code motion accounts for a major part of the reduction.

With the small amount of static experimental results so far, it is too early to draw final conclusions from these numbers. However, the extent of the improvements indicates that there is still significant performance headroom in some tasks which are very fundamental to EPIC: static scheduling and usage of speculation. The optimal schedules show the way.

8 Acknowledgements

This research has been funded by the graduate studies program "Quality Guarantees for Computer Systems" supported by the Deutsche Forschungsgemeinschaft. My further thank goes to the Max-Planck-Institut für Informatik, Saarbrücken, for giving access to their CPLEX installation.

References

[AJLA95] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software Pipelining. *Computing Surveys*, 27(3):367–432, September 1995.

[BFG+00] Robert E. Bixby, Mary Fenelon, Zonghao Gu, Ed Rothberg, and Roland Wunderling. ILOG CPLEX Division. MIP: Theory and Practice - Closing the Gap. In M. J. D. Powell and S. Scholtes, editors, *System Modelling and Optimization: Methods, Theory and Applications*, pages 19–49. Kluwer, The Netherlands, 2000.

[BM00] J. Bharadwaj and C. McKinsey. Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs. *Journal of Instruction-Level Parallelism*, 1(6):1–6, 2000.

[BR91] David Bernstein and Michael Rodeh. Global Instruction Scheduling for Superscalar Machines. *Proceedings of the ACM SIGPLAN '91 on Programming Language Design and Implementation (PLDI)*, June 1991.

[BRS92] D. Bernstein, M. Rodeh, and M. Sagiv. Proving Safety of Speculative Load Instructions at Compile-Time. *Proceedings of the 4th European Symposium on Programming*, 1992.

[CCK97] C-M. Chang, C-M. Chen, and C-T. King. Using Integer Linear Programming for Instruction Scheduling and Register Allocation in Multi-Issue Processors. *Computers and Mathematics with Applications*, 34(9):1–14, November 1997.

[cpl02] ILOG CPLEX 8.0, 2002. www.cplex.com.

[Dan51] G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In Tj. C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. Wiley, New York, 1951.

[DKK+99] Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John

Ng, and David Sehr. An Overview of the Intel® IA-64 Compiler. *Intel Technology Journal*, (Q4), 1999.

[GE93] C.H. Gebotys and M.I. Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1266–1278, 1993.

[GLS01] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)*, pages 47–58, June 2001.

[HB01] Steve Haga and Rajeev Barua. EPIC Instruction Scheduling Based on Optimal Approaches. *Proceedings of the EPIC-1 Workshop*, December 2001.

[HLW00] M. Heffernan, J. Liu, and K. Wilken. Optimal Instruction Scheduling Using Integer Programming. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 121–133, June 2000.

[Int02] Intel. *Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*, June 2002.

[Käs00] Daniel Kästner. *Retargetable Code Optimisation by Integer Linear Programming*. PhD thesis, Saarland University, 2000.

[KW01] Daniel Kästner and Sebastian Winkel. ILP-based Instruction Scheduling for IA-64. *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, June 2001.

[NH02] Samuel D. Naffziger and Gary Hammond. The Implementation of the Next Generation 64b Itanium™ Microprocessor. *Proceedings of the IEEE International Solid-State Circuits Conference*, 2002.

[NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[WGB94] T.C. Wilson, G.W. Grewal, and D.K. Banerji. An ILP Solution for Simultaneous Scheduling, Allocation, and Binding in Multiple Block Synthesis. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 581–586. IEEE Computer Society Press, 1994.

[Win01] Sebastian Winkel. ILP-basierte Instruktionsanordnung für IA-64. Master's thesis, Saarland University, 2001. In German.

Procedure Boundary Elimination for EPIC Compilers

Spyridon Triantafyllis Manish Vachharajani David I. August

Departments of Computer Science and Electrical Engineering
Princeton University
Princeton, NJ 08544
{strianta, manishv, august}@cs.princeton.edu

Abstract

Procedures are the basic units of compilation in traditional optimization frameworks. This presents problems to compilers targeting EPIC architectures, since the limited scope of a single procedure is usually insufficient for extracting ILP and identifying enough optimization opportunities. Although inlining can expand the scope of optimization routines, it is not applicable to all call sites and can cause excessive code growth, which can in turn adversely affect cache performance and compile-time resource usage.

In this paper we propose a novel compilation strategy called Procedure Boundary Elimination (PBE). PBE unifies the whole program into a single compilation unit, which is then restructured into units better suited to optimization than the original procedures. A targeted specialization phase exposes further optimization opportunities while limiting code growth only to the cases where it is beneficial. Unlike inlining, PBE can eliminate all procedure calls while avoiding the cost of excessive code growth.

1. Introduction

Achieving good performance on a modern wide-issue architecture is critically dependent on compiler support. Apart from traditional code simplification and redundancy elimination optimization routines, a modern aggressively optimizing compiler has to efficiently exploit complex computational resources, expose instruction-level parallelism (ILP), and avoid performance pitfalls such as memory stalls and branch misprediction penalties. The dependence of performance on compiler quality is especially pronounced on EPIC architectures, since these architectures require compiler-constructed explicit schedules.

In order to meet the challenges posed by EPIC architectures, a compiler has to rely on a rich set of aggressive optimization and analysis routines. The ability of such routines to produce efficient code can be greatly hampered by the traditional procedure-based compilation approach. This is because the original breakup of a program into procedures serves software engineering rather than optimization goals, and thus individual procedures may not present the best scope to optimization and analysis. For example, procedure calls within loops can conceal cyclic code from the compiler, and breaking up a task into too many small procedures may prevent a scheduling routine from constructing traces long enough to provide sufficient ILP opportunities. Modern software engineering techniques such as object-oriented programming, which typically encourage small procedures and frequent procedure calls, exacerbate the problem.

To alleviate the effects of inconveniently placed procedure boundaries, traditional compilers have employed interprocedural analysis and aggressive inlining. Interprocedural analysis exposes more information to the optimizer and can be employed as extensively as compile time permits. However, designing interprocedural analysis routines is complicated by the fact that such routines have to take into account parameter-passing mechanisms and other calling conventions. Inlining ([1],[2],[3]), originally proposed to limit call overhead, copies the body of a callee procedure into the body of the caller. This not only exposes more code to analysis routines, but also allows subsequent optimization routines to specialize the code of the callee for each particular call site. Unfortunately, the benefits of aggressive inlining come at the cost of extensive code growth. This can lead to poor instruction cache performance, as well as slow down the compilation process. Since the adverse effects of code growth can very quickly become prohibitive, inlining is usually limited to frequently executed call sites with relatively small callees. The applicability of inlining is further limited by its inability to handle recursive and virtual procedure calls. Inlining is therefore only a partial solution to the optimization scope problem.

This paper proposes a novel compilation strategy, called Procedure Boundary Elimination (PBE), to overcome the limitations of traditional procedure-based compilation. The PBE approach first eliminates procedure boundaries without causing any code growth. The program is then partitioned into *regions* [4] in order to present to subsequent optimization routines a set of compilation units that are both manageable in size and adequate in scope. Targeted code specialization is then applied in order to create more optimization opportunities, while limiting code growth only where it is likely to produce significant benefits. Standard optimization and analysis routines, with some essential modifications, are subsequently

applied to each region. In this way PBE offers increased optimization scope and specialization opportunities, while keeping code growth in check. Unlike inlining, PBE handles all call sites in a uniform way, regardless of recursive cycles or callee size. An added benefit of PBE is that the design of global analysis routines is simplified, since such routines do not have to navigate around procedure calls and platform-specific calling conventions.

2. The Optimization Scope Problem

In a traditional compiler each procedure is treated as a separate compilation unit. Since procedures are defined by the programmer according to software engineering considerations, they may not be ideally suited for achieving maximum optimization efficiency. In modern EPIC architectures, where the dependence of performance on aggressive optimization is especially pronounced, limiting the scope of optimization and analysis routines to a single procedure may lead to significant performance degradation. Section 2.1 analyzes this problem in more detail.

To address the problem of performance degradation due to inconveniently placed procedure boundaries, most modern compilers employ aggressive inlining. Although inlining can lead to large performance gains, it is a solution of limited applicability and effectiveness. Section 2.2 presents inlining and explains the need for a more general solution to the problem of optimization scope.

2.1. Problems with Procedure-Based Compilation

In modern software systems the breakup of a large program into smaller, more manageable procedures serves two unrelated and ultimately contradictory purposes. As a programming unit a procedure must be small, elegantly written, and conceptually coherent. As a unit of optimization and analysis a procedure must be large enough to provide an adequately wide scope for optimization and analysis, and should ideally contain pieces of code that are strongly correlated, both in the sense that they usually execute together and in the sense that they operate on common data.

The reason why these two roles may be contradictory is best illustrated through an example. In Figure 1a we can see a small procedure `f` with two arguments, which is called from two different locations: one is in basic block `C`, which is part of a loop `L`, and one is in the less frequently executed block `K`. Hot basic blocks are shown with a thicker border. In the following discussion we will assume that these blocks, that is `B`, `C`, `D`, `E`, `F` and `H`, execute much more frequently than the rest. As for the small pieces of code within the blocks, let `LI1` and `LI2` be two instructions that are invariant with respect to the loop formed by blocks `B`, `C` and `D` Also, let `f(rx, ry)` be an abbreviation for making a call to procedure `f` with parameters `rx` and `ry`; depending on the calling convention, this may correspond to a multiple instruction sequence, including instructions that copy the parameters to specified locations, as well as instructions that save and restore caller-saved registers.

Putting the blocks `E`, `F`, `G` and `H` into a separate procedure may be a good decision from a software engineering perspective. As we can see in Figure 1a, this allows the code of these four blocks to be reused in block `K`. The code in these four blocks may also be conceptually different from that in loop `L`. However, partitioning the code in this way conceals optimization opportunities. Although the code of `f` is frequently executed as part of `L`, it is not part of the loop as far as optimization and analysis are concerned. Thus, loop unrolling or software pipelining would benefit only the three blocks in `L`, despite the fact that executing code in `f` probably accounts for a significant portion of the time spent in `L` during program execution. Moreover, instructions in blocks `C` and `D` cannot be scheduled together. This may limit the ability of the scheduler to exploit instruction-level parallelism.

Classical optimization routines can also suffer from the inconveniently placed procedure boundaries in Figure 1a. For example, although parameter `b` in `f` has the constant value 5, no constant propagation is possible. That's because most dataflow analysis routines operate on the procedure level, and would therefore miss this fact. Also, instruction `LI2` cannot be moved out of the loop `L`, even though it is loop invariant.

Various efforts to remedy this situation have been made in the past. One such effort is to generalize certain analysis routines, so that they can take into account interprocedural information. In the example in Figure 1a, an interprocedural dataflow analysis routine might be able to detect the opportunities for constant propagation in parameter `b`. Although interprocedural analysis can be very useful, it is usually too expensive to be applied extensively.

A much more systematic effort to solve the problems caused by procedure boundaries is aggressive inlining. The benefits and drawbacks of aggressive inlining are presented in the next section.

2.2. Inlining: An incomplete solution

Inlining eliminates inconveniently placed procedure boundaries by duplicating the callee's code into the call site. Although the original purpose of inlining was to eliminate call overhead, in today's optimizing compilers inlining is used aggressively in order to increase the scope of optimization. By making the code of the callee visible to its caller, new optimization opportunities can be identified. At the same time important call sites acquire a private copy of the callee's code, which can then be specialized in that particular context. Inlining has been extensively studied in the literature ([1], [2], [3], [5], [6]).

The small piece of code shown in Figure 1a can be transformed by inlining to the equivalent piece of code shown in Figure 1b. Here the code of `f` has been copied inside the loop `L`. In addition to eliminating the frequently incurred call overhead in block `C`, this creates new optimization opportunities.

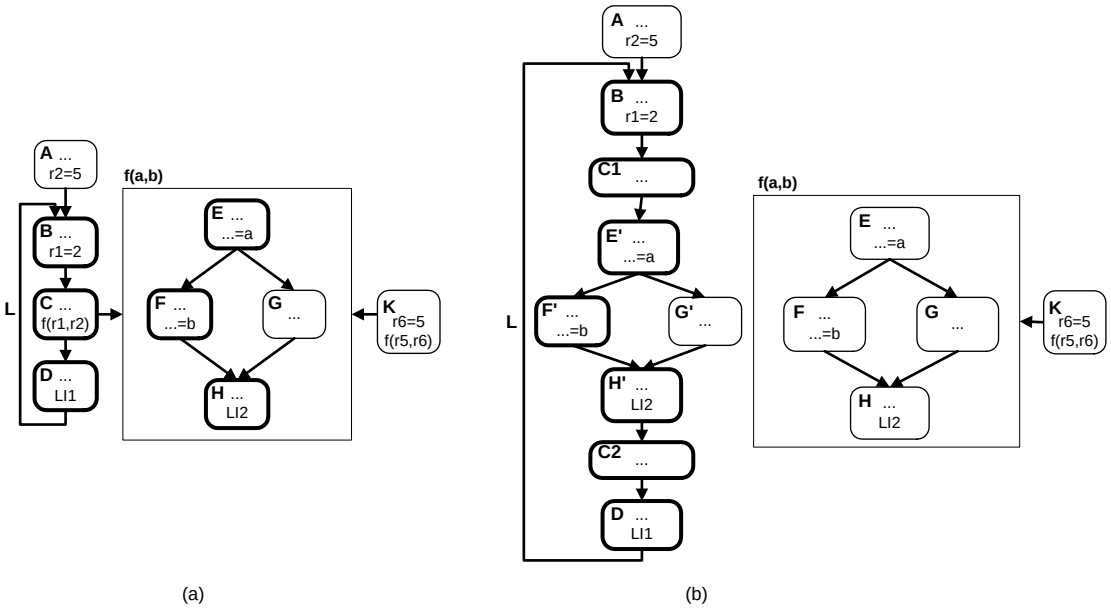


Figure 1: A small procedure with two call sites, (a) before and (b) after inlining

Now instruction `LI2` can be moved out of the loop, and constant propagation can be performed on both `a` and `b`. Loop optimization and scheduling routines can now operate on the whole body of the loop, instead of being limited to blocks `B`, `C`, and `D`.

Unfortunately, the benefits of aggressive inlining come at the cost of extensive code growth. For example, in Figure 1b the body of loop `L` has grown from 3 to 7 basic blocks. Such code growth can adversely affect instruction cache performance, often negating performance benefits due to new optimization opportunities. Moreover, a modern compiler includes many routines of superlinear complexity. When presented with larger procedures as a result of extensive inlining, a compiler's time and memory usage during optimization may grow excessively.

Experimental results found in the literature verify the above observations. For example, the inliner presented in [2] causes an average performance improvement of 11% at the cost of 17% average code growth on a set of eight benchmarks, and [4] reports an increase by 8.3 times in the optimization time of the Perl benchmark when 20% of call sites are inlined.

Code growth concerns limit the applicability of inlining to frequently executed call sites with small callees. Moreover, the steep increase in compile time due to aggressive inlinining forces most compilers to make conservative inlining choices in order to keep optimization tractable. Finally, inlining cannot handle recursive functions properly. Although recursive cycles are in essence loops, inlining cannot expose the looping behavior of recursive functions, which would allow the optimizer to apply loop optimizations to recursive function bodies. Instead, inlining can only eliminate the first stages of recursion, but must ultimately leave the recursive call in place.

The drawbacks of inlining as a solution to the optimization scope problem stem to a great extent from the fact that it was not designed to solve this particular problem. Indeed, although inlining alleviates some of the problems caused by procedure boundaries, it is itself constrained by those boundaries. That is because inlining operates on the procedure level: it can respond to optimization scope problems only by duplicating whole procedures. This leads, among other things, to excessive code growth. In the example in Figure 1b, the whole body of procedure `f` is duplicated, when most optimization benefits are likely to come only from the "hot" path, `E`→`F`→`H`. Partial inlining ([5], [6]) has been proposed to address this problem. However, the freedom of partial inlining to specialize code is still constrained by the original procedure boundaries, the quality of its results depend on the structure of the callee, and it is still not applicable to recursive, virtual, and certain large callees. Thus partial inlining is an effort to alleviate the drawbacks of inlining, rather than an effort to eliminate these drawbacks.

3. Procedure Boundary Elimination

As discussed in section 2, optimization and analysis routines in compilers for modern architectures can suffer from limited optimization scope. Although inlining can extend the scope of optimization routines by copying callee procedures to their call sites, its benefits are offset by significant drawbacks, including excessive code growth and limited applicability.

Our goal in this section is to define a code transformation that eliminates problems caused by inconveniently placed procedure boundaries without causing unnecessary code growth. Such a transformation would obviate the need for inlining,

while at the same time exposing more optimization opportunities and exhibiting better compile-time behavior than inlining. Although some code growth will be allowed, code duplication will only happen when it is deemed beneficial for optimization, and not for the purpose of eliminating procedure calls. We call the proposed method Procedure Boundary Elimination (PBE).

PBE involves three separate phases. The first phase is *procedure unification*, which transforms the whole program into one single compilation unit. In the second phase, *region formation*, this single compilation unit is repartitioned in more manageable pieces using the algorithm proposed in [4]. In the third phase, *targeted code specialization*, selected parts of the program are duplicated in order to provide code specialization opportunities. Apart from these three phases PBE also requires certain modifications in existing optimization and analysis routines in order to be effective.

Procedure unification, region formation and targeted code specialization are covered in Sections 3.1, 3.2, and 3.4 respectively. Required modifications in existing optimization and analysis routines are covered in Section 3.3.

3.1. Procedure Unification

The first step in PBE, called Procedure Unification, is to eliminate procedure boundaries by replacing call and return instructions with normal branches. Thus the whole program effectively becomes a single procedure. As a result, optimization and analysis routines can operate on the widest possible scope. To return to the example in figure 1, the original code will be transformed by procedure unification to the code in figure 2a.

Apart from replacing calls and returns with branches, procedure unification must also take care of the rest of the semantics of a procedure call. These include parameter passing and stack frame setup, as well as saving and restoring caller- and callee-saved registers. We will first present the solution to these problems for nonrecursive procedures, and then we will adapt our solutions to the recursive case.

3.1.1. Parameter passing

PBE is applied early on in the optimization process, before register allocation. Parameter passing can therefore be performed by creating a new virtual register for each parameter of each procedure. Before branching into a (former) procedure body, a piece of code needs to move the procedure's (former) parameters into the virtual registers designated for that procedure. Parameters that are too big to fit into registers can be moved into designated memory locations. Parameter passing for recursive calls requires some extra complication, as discussed in section 3.1.4.

3.1.2. Stack frame setup

Since we are dealing with nonrecursive procedures, there is no need to maintain a stack for activation frames. Instead, the activation frame of each procedure can occupy a constant memory address range into the global variable space. This can be achieved by simply assigning a separate memory address range for the activation frame of each procedure. A more sophisticated implementation can avoid wasting memory space by assigning overlapping memory spaces to procedure stack frames. This is possible, since only procedures that can be active at the same time need non-overlapping stack frames. An interference graph between procedures needs to be created, where two procedures are considered interfering if they can be active at the same time. A graph coloring algorithm can then be used to assign address ranges to activation frames in a near-optimal way.

3.1.3. Handling caller-saved and callee-saved registers

Since we are dealing with nonrecursive procedures, we can just rename virtual registers so that each procedure uses a disjoint range of virtual registers. The register allocation routine can then find an optimal allocation of these virtual registers to actual machine registers.

Forcing each procedure to use a distinct set of virtual registers may cause a huge increase in the number of virtual registers used in a program. However, the increase in register pressure will be much more modest. This is because only virtual registers in procedures that can be active together can interfere with each other. Even then, live range splitting should be able to reduce register pressure in most cases. In the example in figure 2a, a virtual register that is defined in block `B` and used in block `D` increases register pressure in the former body of `f`. However, since such a register is just "live through" blocks `E`, `F`, `G`, and `H`, its live range can be split just before the branch to `E` and restored just after the branch back to `C2`. In the worst case, live range splitting will have to insert as many saves and restores as the original calling convention. However, in most cases live range splitting will be able to exploit its extra degrees of freedom in order to make much better decisions than the original calling convention.

3.1.4. Dealing with recursive procedures

By "recursive procedure" we mean any procedure that participates in a cycle in the original call graph of a program. This covers both simple and mutual recursion. Such procedures can be easily identified before procedure unification.

It is ultimately impossible to implement the semantics of recursive procedure calls without using a stack. Therefore the solutions discussed above for parameter passing, activation frame setup and saving registers are not applicable to recursive

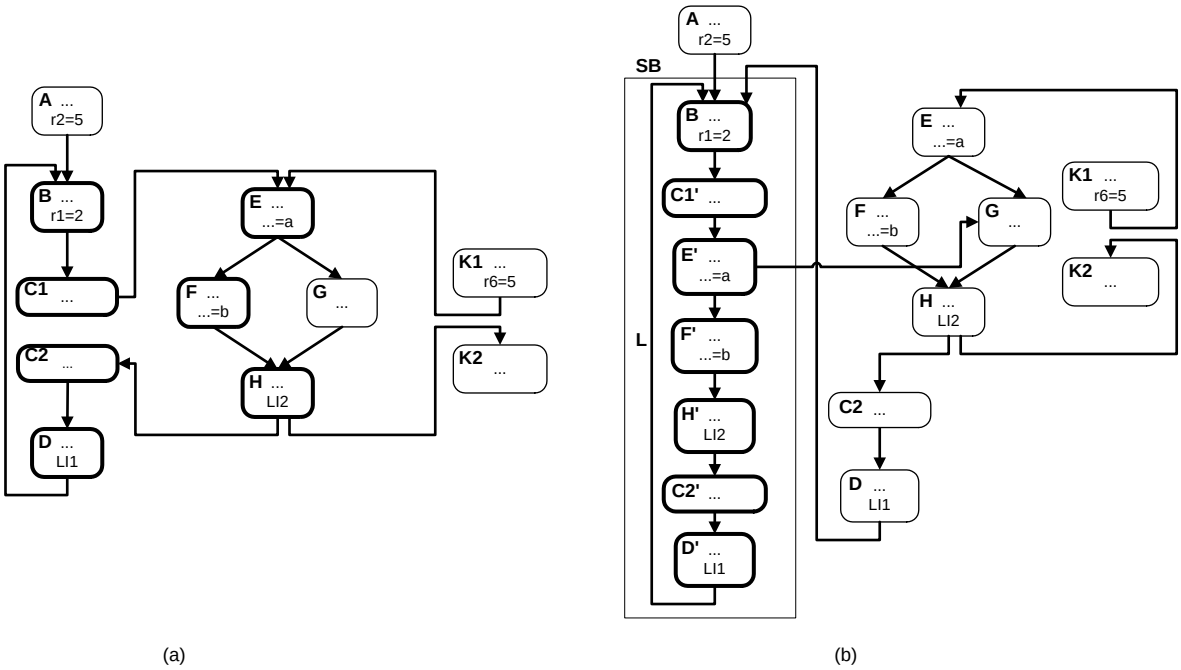


Figure 2: A small procedure with two call sites after (a) procedure unification and (b) procedure unification and superblock formation

procedures. Even after procedure unification, recursive procedures will have to allocate their parameters and local variables in the stack, and save and restore any registers they modify around recursive procedure calls.

However, recursive procedures can also benefit from procedure unification. After procedure unification, recursive call sequences will be transformed to loops. Traditional optimizations can then be applied on those loops. Such optimizations can take advantage of loop invariant code motion and copy propagation in order to reduce the number of registers used within the recursive loop, and therefore the amount of data that has to be saved on the stack.

3.2. Region formation

The result of procedure unification is a single, huge compilation unit. Since aggressive optimizers incorporate many superlinear optimization and analysis routines, a compilation unit of that size is bound to cause an explosion in time and memory usage during optimization. Therefore, PBE cannot be practical unless this compilation unit can be repartitioned into more manageable pieces.

Fortunately a method for partitioning large compilation units into smaller, more manageable pieces has already been proposed. This method, called *region formation*, has been described in detail in [4]. Region formation has been originally developed in order to cope with the compile-time explosion caused by over-aggressive inlining, but it becomes even more valuable in the context of PBE.

Region formation uses profile information in order to break up a large compilation unit into *regions*, i.e. pieces of code that usually execute together. These regions are subsequently used as units of compilation. Experimental results in [4] demonstrate that region formation can keep optimization time constant in the presence of ever-growing procedure size, at the cost of insignificant performance penalties at runtime. Although these experimental results refer to inlining, there is no reason to assume that region formation cannot be just as effective in the context of PBE.

3.3. Modifications in existing optimization and analysis routines

Expanding the scope of optimization and analysis routines cannot be accomplished by simply eliminating procedure boundaries. This is because procedure unification introduces irregular program structures that existing compiler routines may not be able to deal with.

For example, as shown in figure 2a, the natural loop `L` is lost after procedure unification. That is because blocks `E`, `F`, `G`, and `H` are not dominated by the loop's head, `B`. Moreover, a traditional dominator analysis routine would conclude that blocks `C2` and `D` can be reached through the path `K1`→`E`→`F`→`H`→`C2`→`D`, although this path is in fact never taken. Therefore loop optimizations are not available on these blocks. For example, although the loop-invariant instruction `LI1` can be safely moved in block `A`, traditional analysis routines would conclude that such a move is not legal.

The solution lies in realizing the relation between call and return arcs. In the example of figure 2a, the definition of dom-

inator analysis has to be modified in order to account for the fact that the arcs C1→E and H→C2 are always executed together. Also, arc C1→E can never be followed by arc H→K2, and arc H→C2 can never be preceded by arc K1→E. Using this fact, a modified dominator analysis routine can conclude that block D is dominated by block B. This in turn makes loop optimizations available on blocks B, C1, C2, and D. For example, instruction LI1 can now be moved out of the loop using loop invariant code motion. On the contrary, instruction LI2 cannot be moved, since it is shared between the loop and blocks K1-K2.

Every analysis and optimization routine in the compiler has to be modified in a similar fashion, in order to take into account call and return arc relations.

3.4. Targeted code specialization

Although the modifications described above increase the optimizer's ability to handle the irregular control flow graphs produced by procedure unification, the resulting optimizer still misses many opportunities that were available during inlining. The reason is that inlining, apart from eliminating calls, also makes a private copy of the callee available for specialization.

Instead of duplicating whole procedure bodies, PBE relies on a *targeted code specialization* phase. A targeted specialization routine can duplicate only selected parts of the program, without being constrained by the original procedure breakup. Therefore it is much less likely than inlining to cause unnecessary code growth. Such a code specialization routine can also benefit all parts of the program, even the ones that don't involve procedure calls.

The first code specialization routine we tried in the context of PBE is an implementation of *superblock formation* that is aware of call and return arc relations. As described in [7], superblock formation identifies hot paths through the code, and then uses tail duplication in order to eliminate side entrances to these paths. Thus each such path can become a single, straight-line "superblock". In our simple example, superblock formation will choose to form a superblock out of the hot path B→C1→E→F→H→C2→D, resulting to the code in Figure 2b.

One can easily see that all optimization opportunities identified in Section 2.2 also exist in the code of Figure 2b. Notice however that the "cold" block G has not been duplicated. Since it is well known that only a small portion of each procedure's code is hot, this difference can lead to significantly less code growth in comparison to inlining.

Although superblock formation is a valuable technique in the context of PBE, a more sophisticated code specialization routine is needed. One of the drawbacks of superblock formation in the context of code specialization is that it can only duplicate a single path through a hot code segment. In many cases making several paths simultaneously available for specialization may be desired. The fact that superblock formation makes its choices relying only on execution weight data is also a drawback. A more sophisticated code duplication method should be able to rule out cases where code duplication does not generate specialization opportunities. Such a method should also be able to duplicate selected portions of the control-flow graph, even portions that do not constitute straight-line paths. We are in the process of developing such a code specialization routine for PBE.

4. Preliminary experimental results

In order to perform a preliminary evaluation of the PBE concept, we used a simple experimental setup. In it, we used the IMPACT compiler [8], in conjunction with procedure unification and superblock formation routines[1] implemented in the Liberty compiler [9]. Since region formation was not available, this system could only handle small benchmarks due to compile time limitations.

Figure 3 compares the results of PBE with those of inlining on three benchmarks: 129.compress from the SPEC'95 benchmark suite, and the UNIX utilities grep and yacc.

Results for inlining were obtained using IMPACT's default inlining routines. For PBE each benchmark was compiled in IMPACT up to a low-level intermediate representation (Lcode). This involved some high-level optimization, but no inlining. The intermediate representation was then saved to a file and passed on to the Liberty compiler for procedure unification and superblock formation. The resulting intermediate representation was fed back to IMPACT for low-level optimization and scheduling.

The first column of the graph in Figure 3 shows the code size of the executables resulting from PBE as a percentage of the code size of the corresponding executables obtained using inlining. The second column does the same for dynamic cycle count. Dynamic cycle counts were obtained by scheduling the programs using IMPACT's scheduling routines for a hypothetical 8-issue EPIC machine, and estimating their cycle counts using profile data, without taking into account cache misses or branch prediction behavior.

As we can see, PBE on average reduces the cycles executed by 16.8%, while producing executables that are 5.7% smaller. The improvements in runtime performance are in fact understated, since our preliminary experiment does not account for the improvement in instruction cache behavior resulting from less code growth. Implementing a better code specialization routine is likely to reduce code growth, improve performance, or both.

5. Conclusion

In this paper we discuss how procedure-centric compiler design methodologies limit the potential of EPIC architectures by limiting the optimization scope available. We then discuss

[1] IMPACT has its own superblock formation routine, but that routine was disabled for the purposes of this experiment.

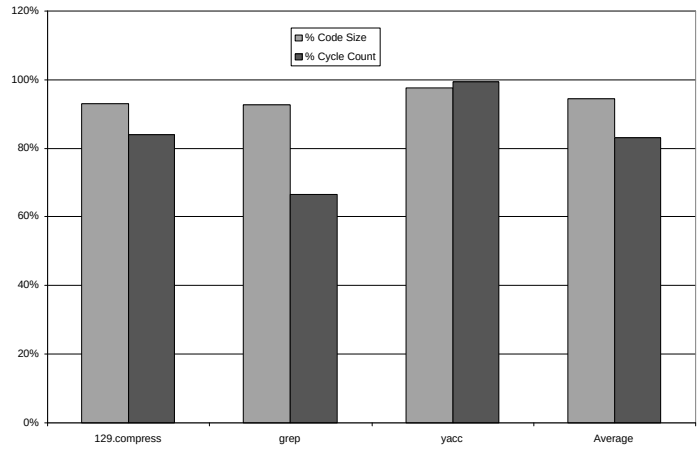


Figure 3: Comparison between inlining and PBE in terms of code growth (a) and static cycle count (b)

how inlining can resolve the issues with optimization scope but only at the cost of code-size. We then propose a new compilation strategy, Procedure Boundary Elimination (PBE) that eliminates procedure boundaries early in the compilation cycle, without code growth. PBE then selects new compilation units via region formation. Since the compiler selects these compilation units, they are selected for optimization potential, and not software engineering concerns as is the case with program procedures. To recover the benefits of code specialization provided by code duplication in inlining, PBE uses a targeted code specialization technique to duplicate code, but only code that will result in performance improvement, instead of entire procedures. In this way PBE achieves all the benefits of inlining without the code size growth and associated performance penalties. After proposing PBE, we discuss the key challenges in building a compiler that uses this technique.

By rethinking the overall strategy that the compiler uses to select compilation units, it should be possible to dramatically improve the performance of EPIC architectures. The preliminary results obtained by implementing only a limited version of PBE are promising, motivating the effort necessary to construct a compiler with full PBE support. We expect that such an implementation of PBE will prove valuable in compilation for EPIC architectures.

References

[1] W. W. Hwu and P. P. Chang, "Inline function expansion for compiling C programs," in *Proceedings of the '89 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 246–257, June 1989.

[2] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Profile-guided automatic inline expansion for C programs," *Software Practice and Experience*, vol. 22, pp. 349–370, May 1992.

[3] A. Ayers, R. Schooler, and R. Gottlieb, "Aggressive inlining," in *Proceedings of the '97 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 134–145, June 1997.

[4] R. E. Hank, W. W. Hwu, and B. R. Rau, "Region-based compilation: An introduction and motivation," *International Journal of Parallel Programming*, vol. 25, pp. 113–146, April 1997.

[5] T. Way, B. Breech, and L. Pollock, "Region formation analysis with demand-driven inlining for region-based optimization," in *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pp. 24–33, October 2000.

[6] T. Way and L. L. Pollock, "A region-based partial inlining algorithm for an ILP optimizing compiler," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, June 2002.

[7] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

[8] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predication and speculative execution in the IMPACT EPIC architecture," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 227–237, June 1998.

[9] "The Liberty Project Website." http://liberty.princeton.edu/.

Predicate Analysis and If-Conversion in an Itanium Link-Time Optimizer *

Noah Snavely, Saumya Debray, Gregory Andrews

Department of Computer Science
University of Arizona
Tucson, AZ 85721.

{snavely, debray, greg}@cs.arizona.edu

ABSTRACT

EPIC architectures, such as the Intel IA-64 (Itanium), combine explicit instruction-level parallelism with instruction predication. To generate efficient code, it is important to use predication effectively. In particular, it is important to replace conditional branches and multiple code blocks by single, branch-free code blocks when doing so would lead to faster code. This process, which is known as if-conversion, is generally done early in the code-generation process; hence subsequent analyses and optimizations have to deal with predicated code. This paper examines an alternative approach in which code is unpredicated during disassembly, the internal representations are virtually identical to those in a conventional architecture (specifically the IA-32 Pentium) and if-conversion is done late in the compilation process, at the same time as instruction scheduling and just before code layout. This paper also presents new algorithms for analyzing predicated code and evaluates their efficacy. We show that our approach is able to produce code that is denser (fewer nop instructions) and almost as fast as the best code produced by the Intel `ecc` compiler on the SPECint-2000 benchmark suite. On the same programs, our predicate analysis and if-conversion algorithms lead to an average speed improvement of a little under 6% on the best code produced by the `gcc` compiler.

1. INTRODUCTION

There has been a great deal of recent interest in EPIC (Explicitly Parallel Instruction Computing) architectures, such as the Intel IA-64 (Itanium), that support predicated instructions and explicit instruction-level parallelism. A predicated instruction is guarded by a Boolean source operand; the instruction is executed only if this guard evaluates to true. In addition, instruction-level parallelism is explicit: the compiler is responsible for collecting instructions into groups that will be executed in parallel.

In order to make effective use of the capabilities of such architectures, a compiler must selectively eliminate conditional jumps in favor of predicated instructions that are conditionally executed. This process, known as *if-conversion*, must be carried out judiciously: if it is too aggressive, it leads to contention for system resources and a concomitant degradation in performance; if it is not aggressive enough, it results in insufficient instruction-level parallelism, which also leads to a loss in performance. There are two important questions that have to be addressed in this regard. The first is that of when if-conversion should be carried out in the compilation process. The second is that of determining relationships between predicates, which is necessary to identify dependencies between predicated instructions.

One option for carrying out if-conversion is to do it early in the code generation process, with subsequent analyses and optimizations working on predicated code. This is the approach taken by August *et al.* [3], who carry out aggressive if-conversion early, and subsequently perform partial reverse if-conversion during instruction scheduling. The advantage of such an approach is that the compiler can take full advantage of instruction predication in a variety of low-level optimizations. A disadvantage of such an approach is that analyses and optimizations in the compiler backend may have to be reimplemented to cope with predication. An alternative is to delay if-conversion until much later in the compilation process, during instruction scheduling, after most optimizations have been carried out. The advantage here is that other analyses and optimizations do not have to be made predication-aware. This can simplify the construction of portable multi-target optimizers.

The determination of relationships between predicates can be helpful for improving the quality of code generated. For example, during instruction scheduling, it can allow the compiler to exclude false scheduling dependencies between instructions whose guarding predicates cannot be simultaneously true. Another situation where knowledge of such relationships can be useful is in the context of profile-guided code layout, which can enhance program performance by improving its instruction cache behavior [12]. This can sometimes require us to invert the sense of a branch, e.g., so as to have it fall through rather than be taken. If the branch instruction in question is conditioned on a predicate register p, and we know that a predicate q is guaranteed to be the complement of p at that point, we can achieve this inversion simply by replacing the guard predicate p by its complement q. Without such knowledge, we may have to explicitly compute the complement of p, which can take several instructions and adversely affect performance.

The contribution of this paper is to present algorithms for if-conversion and predicate analysis and to evaluate their efficacy

*This work was supported in part by the National Science Foundation under grants CCR-0073394, EIA-0080123, and CCR-0113633.

experimentally. Our approach is very different from those that have been discussed in the literature. To simplify implementation and keep analyses and optimizations architecture-independent, our internal representation does not support predicated instructions. We therefore carry out reverse if-conversion after disassembly, converting instructions into their unpredicated form and introducing conditional branches in the process, following which a conventional control flow graph is constructed. This control flow graph is then subjected to various analyses and optimizations in the usual way. We then carry out if-conversion during instruction scheduling, just prior to code layout. Our approach can be seen as dual to that of August *et al.* [3]. Our predicate analysis is also quite different from previous proposals for determining relationships between predicates: we use a dataflow analysis that is able to handle arbitrary control flow, and can be extended in a straightforward way to an inter-procedural analysis.

The remainder of the paper is organized as follows. Section 2 gives background information on the Itanium architecture and on our experimental optimization system. Section 3 describes how we analyze the use of predicate registers to compute what we call weak and strong disjointness sets. Section 4 presents our if-conversion algorithm. Section 5 contains examples that illustrate the use of predicate analysis and if-conversion. Section 6 address two questions: How effective is our approach in improving a mostly unpredicated instruction stream? And how effective is it in identifying opportunities for if-conversion? That section describes our experimental method and shows that the answers to both questions are positive. Finally, Section 7 discusses related work, and Section 8 gives concluding remarks.

2. OVERVIEW

The work reported in this paper was carried out in the context of ILTO, a link-time optimizer we have developed for the Intel Itanium processor. This section summarizes relevant aspects of the Itanium architecture, including predicated instructions, instructions groups, bundles, and templates. Then we give an overview of the processing stages in ILTO and identify the places where it employs predicate register analysis.

2.1 Itanium Architecture

The Itanium contains multiple functional units and uses programmer specified instruction-level parallelism. Moreover, every instruction is *predicated*: It specifies a one-bit predicate register, and if the value of that register is true (1), then the instruction is executed; otherwise, the instruction usually has no effect. The Itanium has 64 predicate registers; register `p0` has constant value true (assignments to it are ignored). Many instructions in programs use `p0` as their predicate; these are said to be *unguarded* and by convention the predicate register is not specified in assembly code (as shown below). Instructions that specify a predicate register other than `p0` are said to be *guarded*.

Predicate registers are set by compare instructions. There are three broad classes of compares: normal, unconditional, and parallel. A normal compare has four operands: two data operands that are compared, and two predicate registers that are assigned the result and its complement. An unconditional compare is like a normal compare, except that it clears both predicate-register operands before doing the data comparison and setting the results; moreover, the predicate registers are cleared even if the instruction is not executed because its guard is false. A parallel-OR compare sets both predicate-register operands if the data comparison is true; otherwise neither predicate register is changed. A parallel-AND compare clears both predicate-register operands if the data comparison is false; otherwise neither predicate register is changed. Parallel compares are used to compute sequences of logical OR and logical AND operations.

The compiler writer or assembly programmer expresses parallelism by forming what are called *instruction groups*. Each group is a sequence of instructions that do not contain register dependencies and hence that can potentially be issued in parallel. In particular, instructions in a group cannot in general contain read-after-write (RAW) or write-after-write (WAW) register dependencies. (Write-after-read dependencies are allowed in a group since the processor will ensure that the read occurs before the data is overwritten.) The programmer indicates the end of an instruction group by means of what are called *stop bits*.

Following is an example of a sequence of predicated instructions:

```
        cmp.eq p6,p7=r10,r11
(p6)    ld8    r15=[r32]
(p7)    ld8    r16=[r33] ;;
(p6)    add    r15=r15,1
(p7)    add    r16=r16,1 ;;
(p6)    st8    [r32],r15
(p7)    st8    [r33],r16
```

The first instruction is unguarded and always executed. It compares the contents of general registers `r10` and `r11`; if they are the same, predicate register `p6` is set to true and register `p7` is set to false; otherwise `p7` is set to true and `p6` is set to false. Because the values of `p6` and `p7` are complements of each other, exactly one set of load, add, store instructions will execute, depending on which of `p6` or `p7` is true. There are register dependencies between the add and load instructions, and between the store and add instructions, so stop bits—indicated by double semicolons `;;`—are placed after the pair of loads and the pair of adds.

The Itanium processor fetches instruction *bundles* that are 128 bits long (two words). Each bundle consists of three 41-bit instruction *slots* and a 5-bit *template*. The template specifies the kind of functional unit needed by each instruction—integer, memory, branch, etc.—and where stop bits are located. The processor views up to two bundles (six instructions) at a time and attempts to *disperse* all of them to functional units in parallel. An instruction can be dispersed when a functional unit is available; up to six instructions can be dispersed at the same time, but instructions are never dispersed out of order.

An instruction *issues* when it can be dispersed and when all the resources it requires (e.g., source registers) are available. A *split issue* occurs whenever an instruction does not issue at the same time as the previous instruction. (Split issue leads to a delay of at least one clock cycle.) Stop bits always cause a split issue, because they indicate the presence of register dependencies. On the other hand, predication never causes a split issue.

To summarize, Itanium instructions are predicated, and they have to be placed into groups (demarcated by stop bits) and bundles (with associated templates). Using predicates wisely and scheduling instructions efficiently are thus keys to producing efficient code.

2.2 ILTO: Itanium Link-Time Optimizer

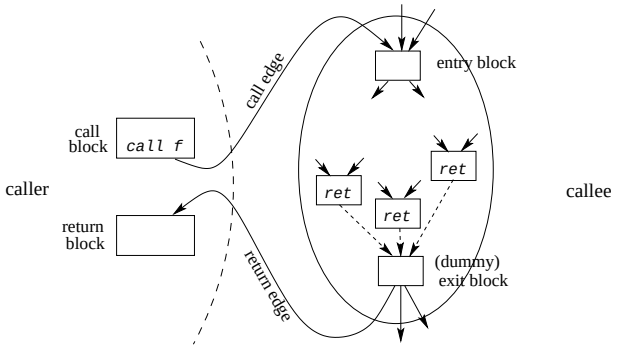


Figure 1: Representing function calls in the interprocedural control flow graph

Our experimental infrastructure is a software system called ILTO (Itanium Link-Time Optimizer). ILTO has the same basic structure as PLTO, a link-time optimizer we have developed for the Intel IA-32 (Pentium) architecture [13]. In particular, ILTO reads in a binary object file, disassembles the code, carries out numerous analyses and code optimizations, performs if-conversion and code scheduling, and finally lays out code blocks and assembles a new binary.

For code analysis and optimization purposes, ILTO constructs a control flow graph (CFG) for each function in a program [1]. Control flow across function boundaries is represented using an *interprocedural control flow graph* (e.g., see [11]). It consists of the control flow graphs of all the functions in the program, together with edges representing calls and returns that connect the flow graphs of different functions. As shown in Figure 1, a function call is represented using a pair of blocks, a *call block* and a *return block*. There is a *call edge* from a call block to the entry block of the callee, with a corresponding *return edge* from the exit block of the callee to the return block. Indirect function calls are modelled using a special pseudo-function $\mathsf{F}_{\perp}$ that represents worst-case behaviors; e.g., it uses and defines all registers, writes to all memory locations, etc.

Disassembly and assembly are obviously architecture dependent. However, the representation of basic blocks, structure of the CFG, and—most importantly—the various analyses and optimizations are essentially the same as in PLTO. The special characteristics of the Itanium—such as predication, instruction groups, and bundles—are thrown away as the control flow graph is created. This lessened the time it took to develop ILTO, and more importantly it permits existing architecture-independent analyses and optimizations to be employed. However, it means that we have to deal with predication, stop bits, and bundling when scheduling and laying out code.

The ILTO system has nine major stages:

1. *Build Control Flow Graph.* Disassemble instruction bundles and build a control flow graph (CFG) with individual instructions. Eliminate dead code by doing a depth-first search from the entry point to mark reachable code.

2. *Predicate Analysis.* Compute predicate register disjointness sets, as described in Section 3.

3. *Unpredicate the CFG.* Remove predication from the instructions in the CFG by constructing explicit decision nodes.

4. *Code Optimizations.* Analyze and optimize the code: liveness analysis, function inlining, constant propagation, etc. For this paper, this phase is not used, as discussed in the results section.

5. *Predicate Analysis.* Recompute predicate register disjointness sets.

6. *Scheduling and If-Conversion.* Form a schedule for each basic block and convert decision nodes to predicated instructions where possible. Group instructions into bundles.

7. *Predicate Analysis.* Recompute predicate register disjointness sets.

8. *Code Layout.* Layout and align the basic blocks, using edge profiles as a guide. (Edge profiles are generated during a training run on an instrumented version of the unpredicated CFG.)

9. *Global Bundle Check and Patch.* Iterate through the basic blocks to check the validity of instruction bundles and to repair them when needed.

Predicate disjointness sets specify relations between the values of predicate registers at the start of each basic block. They are consulted as the CFG is unpredicated in order to simplify some cases, used extensively during if-conversion and instruction scheduling to produce good code, and used when blocks are moved during code layout in order to change the sense of branches. The definition, construction, and use of predicate disjointness sets are described in the next section.

3. PREDICATE ANALYSIS

Given Booleans p and q, '$p \Rightarrow q$' denotes logical implication, i.e., $p \Rightarrow q \equiv (\neg p) \vee q$, while '$p \Leftrightarrow q$' denotes logical equivalence, i.e., $p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$. We define the following notions of disjointness:

DEFINITION 3.1. Booleans p and q are said to be *weakly disjoint* if $p \Rightarrow \neg q$. They are said to be *strongly disjoint* if $p \Leftrightarrow \neg q$. ∎

Note that both weak and strong disjointness are symmetric—e.g., if $p \Rightarrow \neg q$, then $q \Rightarrow \neg p$—so it is not necessary to specify directionality for either of them.

As an example, the following instruction sets predicate registers `p6` and `p7` to complementary values, depending on whether general register `r5` is less than register `r6`:

```
cmp.lt p6,p7=r5,r6
```

Immediately after this instruction, `p6` and `p7` are strongly disjoint, independent of their actual values. They remain strongly disjoint until some instruction (on some path) invalidates the relationship.

Suppose the next instruction that alters `p6` or `p7` is

```
initialize weakIN(B) and strongIN(B) as described in the text;
weakOUT(B) = weakIN(B) ; strongOUT(B) = strongIN(B) ;
for each instruction I in basic block B in their order of occurrence in B do
  if I is not a compare instruction then continue;
  /* Assume I has the form: (pG) compare-opcode pA,pB=data-operands */
  if I is a normal compare instruction then
      if I is unguarded, i.e., pG == p0 then
          remove all pairs containing pA or pB from weakOUT(B) and strongOUT(B) ;
          add (pA, pB) to weakOUT(B) and strongOUT(B) ;
      else
          set wasInWeak to true if (pA, pB) is in weakOUT(B) and to false otherwise;
          set wasInStrong to true if (pA, pB) is in strongOUT(B) and to false otherwise;
          remove all pairs containing pA or pB from weakOUT(B) and strongOUT(B) ;
          if wasInStrong then
              add (pA,pB) to weakOUT(B) and strongOUT(B) ;
          else if wasInWeak or pG == pA or pG == pB then
              add (pA,pB) to weakOUT(B) ;
          else
              /* now no relations between pA and pB */
          end if
      end if
  else if I is an unconditional compare instruction then
      remove all pairs containing pA or pB from weakOUT(B) and strongOUT(B) ;
      add (pA, pB) to weakOUT(B) ;
      for all (p, pG) that are in weakOUT(B) do
          add (p, pA) and (p, pB) to weakOUT(B) ;
      end for
  else /* I is a parallel AND or OR compare instruction */
      remove all pairs containing pA or pB from weakOUT(B) and strongOUT(B) ;
  end if
end for
```

Figure 2: Computing Predicate Disjointness Sets for a Basic Block

```
(p8) cmp.eq p6,p7=r10,r11
```

This instruction is executed conditionally, depending on whether p8 is true. However, p6 and p7 will still be strongly disjoint, even though their values might have changed. (If p6 and p7 were weakly disjoint before this instruction, they would also be weakly disjoint after it; if we knew nothing about their relation before the instruction, we would still know nothing.)

The weakly disjoint relationship most often arises due to instances of unconditional compare instructions. An example is

```
(p8) cmp.unc.eq p6,p7=r10,r11
```

If p8 is true, the semantics of this instruction are the same as a normal compare. However, an unconditional compare first clears both predicate operands, p6 and p7 above, and these remain cleared if the guard predicate is false. Thus, after this instruction, p6 and p7 are weakly disjoint: they cannot both be true but they might both be false.

In order to do effective if-conversion and instruction scheduling (see Sections 4 and 5), we need to know—at each instruction—how predicate registers are related to each other. In particular, for a given register, which other register is strongly disjoint from it, and which other registers are weakly disjoint from it? (There can be at most one register that is strongly disjoint, but there could be several that are weakly disjoint.) The predicate analysis phases in the ILTO system compute this information for the start and end of each basic block in a program, as described below. (It is straightforward to propagate information from the start of a basic block to instructions in the block.)

Our predicate analysis is a forward dataflow analysis that propagates sets of pairs of predicates (p, q) over the control flow graph of a function. We consider two kinds of such sets at each basic block B:

DEFINITION 3.2. Set $weakIN(B)$ is the set of pairs of weakly disjoint predicates at the entry to block B, and $weakOUT(B)$ is the set of pairs of weakly disjoint predicates at the exit from block B. Similarly, $strongIN(B)$ is the set of pairs of strongly disjoint predicates at the entry to block B, and $strongOUT(B)$ is the set of pairs of strongly disjoint predicates at the exit from B. ∎

Let B_0 denote the entry block of the function under consideration. The following dataflow equations specify how the above four sets are computed.

1. The dataflow information at the exit from a basic block B is obtained, as usual, by taking the dataflow information entering B and propagating it through B. In particular, $weakOUT(B)$ is a function of $weakIN(B)$ and the in-

structions in B, and similarly $strong\mathsf{OUT}(B)$ is a function of $strong\mathsf{IN}(B)$ and the instructions in B.

2. Determining disjointness relationships at the entry to a block B involves three cases:

   (a) For intraprocedural analysis we assume that nothing is known at the entry block B_0 to a function:
   $$weak\mathsf{IN}(B_0) = strong\mathsf{IN}(B_0) = \emptyset.$$

   (b) If B is the return block for a call to a function f from a block B', then the dataflow information entering B is obtained by taking the disjointness relations that hold at exit from B', i.e., just before control is transferred to f, and filtering this through the summary information known about the behavior of the callee function f:
   $$weak\mathsf{IN}(B) = \mathsf{FnOut}_f(weak\mathsf{OUT}(B')),$$
   and
   $$strong\mathsf{IN}(B) = \mathsf{FnOut}_f(strong\mathsf{OUT}(B')).$$

   (c) Otherwise, it consists of the disjointness relations that hold at the exit from each of B's predecessors, and so are guaranteed to hold at entry to B:
   $$weak\mathsf{IN}(B) = \bigcap_{P \in preds(B)} weak\mathsf{OUT}(P),$$
   and
   $$strong\mathsf{IN}(B) = \bigcap_{P \in preds(B)} strong\mathsf{OUT}(P).$$

Figure 2 gives the algorithm for computing $weak\mathsf{OUT}(B)$ and $strong\mathsf{OUT}(B)$ from $weak\mathsf{IN}(B)$ and $strong\mathsf{IN}(B)$. There are several cases to consider, but the details are straightforward applications of the kinds of reasoning illustrated in the examples at the start of this section. For example, a normal comparison makes its predicate-register operands strongly disjoint and hence also weakly disjoint; thus, the pair of operands gets added to both the strong and weak output sets. The unconditional compare instruction has the most complex effect, because it clears both predicate-register operands before conditionally setting one of them. A parallel compare instruction has the simplest effect with respect to predicate disjointness because it either does nothing or modifies both predicate-register operands, and hence it destroys any disjointness relationship that might have existed for either predicate register.

We solve the dataflow equations given above by starting with the initial values

$$weak\mathsf{IN}(B) = strong\mathsf{IN}(B) = \emptyset$$

$$weak\mathsf{OUT}(B) = strong\mathsf{OUT}(B) = \emptyset$$

for all basic blocks B in a function, and then computing a fixpoint by iteratively applying the equations above until there is no change to any of these sets.

In case 2(b) of the dataflow equations above, $\mathsf{FnOut}_f(S)$ denotes the effect of the function call f on the disjointness relations at the call site. A simple conservative estimate for intraprocedural analyses is to assume that nothing is known about disjointness relationships at the return from a function call. We can do better, however, by identifying for each function f, the set $\mathsf{Unchg}(f)$ of predicate registers whose values will not be affected by a call to f. We proceed as follows:

1. Define $\mathsf{SaveRestore}(f)$ to be the set of predicate registers that are saved at entry to f before any use, and restored prior to leaving f. These sets can be determined by inspecting the prolog and epilog of f's code.

2. Let $\mathsf{Unchg}(B)$ be the set of predicate registers whose values will not be changed during the execution of B:
   $$\mathsf{Unchg}(B) = \begin{cases} \emptyset & \text{if } B \text{ ends in a function call} \\ \{p \mid p \text{ not assigned to in } B\} & \text{otherwise} \end{cases}$$

Then, the set of predicate registers that are unaffected by a call to f is given by

$$\mathsf{Unchg}(f) = \mathsf{SaveRestore}(f) \bigcup \left(\bigcap_{B \in blocks(f)} \mathsf{Unchg}(B) \right).$$

Note that the set $\mathsf{Unchg}(f)$ can be computed in a single pass over the instructions of f. We can then define the effect of a call to a function f on predicate disjointness relationships as follows:

$$\mathsf{FnOut}_f(S) = \{(p,q) \in S \mid \{p,q\} \subseteq \mathsf{Unchg}(f)\}.$$

This is a pessimistic estimate of the effects of a function call, because when computing $\mathsf{Unchg}(B)$ for a basic block B, we assume that all predicate registers may be overwritten if B contains a function call. A better approach is to propagate $\mathsf{Unchg}(f)$ values over the call graph of the program and iterate to a fixpoint. This is what we have implemented.

It is relatively straightforward to extend these equations to do inter-procedural analysis. At this time, we have extended the analysis described above into a simple context-insensitive inter-procedural algorithm, and we are looking into a context-sensitive inter-procedural version.

4. IF-CONVERSION

If-conversion is the process of replacing explicit control transfers in code by predicated instructions that are executed conditionally depending on the value of a Boolean source operand [2]. It can improve performance in a number of different ways. First, it can eliminate difficult-to-predict branches and reduce branch misprediction rates [4]. Second, it can increase instruction-level parallelism. Finally, by allowing the producer of a value to be moved to an earlier point in the instruction stream, if-conversion can be used to hide instruction latencies.

Figure 3 gives an outline of our if-conversion algorithm. The basic idea is simple: For each basic block in a function, we first schedule the instructions in the block, then we try to use if-conversion to improve the code for that block. This employs the predicate disjointness sets described in the previous section and is done as follows:

1. We attempt to replace nops in the block by useful instructions from its successor blocks.

2. If a block ends in a conditional branch, and it is profitable and possible to eliminate this branch, we replace the conditional branch by appropriately predicated instructions from the block's successors.

In this context, given a basic block B and a successor B' of B, we say that B' is *if-convertible into* B if every instruction in B' can

```
for each basic block B in the function do
  1. schedule B;
  2. sort the successors of B in decreasing order of execution frequency;
  3. for each successor S of B do
       if S has more than one predecessor continue;
       for each nop N in B do     /* Eliminate no-ops in B if possible */
         if there is an instruction I in S that can replace N without affecting any
                  dependencies or adding stop bits then
           remove I from S;
           replace N with an appropriately predicated version of I;
         endif
       end for
       /* Eliminate branch instructions in B if possible and profitable */
       if (a) S is if-convertible into B; and (b) there is a branch instruction J in B that
             can be eliminated by fully if-converting S into B; and (c) the number of
             groups in S is less than a fixed [architecture-dependent] threshold then
         replace each instruction K in S by an appropriately predicated version of K in B;
         delete the branch instruction J
         delete the basic block S
       end if
     end for
end for
```

Figure 3: The Basic If-Conversion Algorithm

be if-converted into a predicated version that can then be inserted at the end of B, prior to any branch instruction at the end of B, without altering any use-definition relationships between any pair of instructions.

A few aspects of this algorithm that deserve comment. First, when processing a basic block B and considering a successor block from which to if-convert instructions into B, we do not consider any successor S that has more than one predecessor. The reason for this is that if S has multiple predecessors, then each instruction moved out of S would have to be replicated in the predecessors of S. This would result in code growth, and it would complicate the if-conversion algorithm because it would be necessary to ensure that such code replication preserves correctness. In principle we could clone the block S in such circumstances to create a block with a single predecessor, which can then be processed as described; however, our implementation does not currently do this.

Second, when considering whether to use if-conversion to eliminate a branch instruction at the end of a block B, we want to make sure that this does not introduce so many predicated instructions into B that the cost of executing these instructions exceeds the cost of the original branch instruction they replaced. We do this using an architecture-dependent threshold that models the cost of executing a branch instruction: if the number of predicated instruction groups being introduced into B is less than this threshold, it is deemed profitable to eliminate the branch instruction. The reason we first attempt to use instructions from S to eliminate no-ops in B before attempting to eliminate branch instructions in B is that the number of instructions in S may initially exceed this threshold, but by pulling out instructions from S to replace no-ops in B, we may be able to reduce the number of instructions in S to below the threshold, thereby allowing the branch instruction in B to be eliminated.

Finally, an aspect of the overall if-conversion process that is not discussed in Figure 3 is that it is sometimes necessary to find a free predicate register. Consider the following code fragment:

```
        cmp.eq p6,p0=r14,r15 ;;
  (p6)  br.cond L1
        mov r14=0
        br.few L2 ;;
L1:     mov r14=1 ;;
L2:     add r15=r14,2
```

We would like to convert this to a single predicated block, e.g.:

```
        cmp.eq p6,p7=r14,r15 ;;
  (p6)  mov r14=0
  (p7)  mov r14=1 ;;
        add r15=r14,2
```

However, since the compare instruction that sets register `p6` in the original code discards the complement of `p6`,[1] we must find a predicate register to hold the complement. This register p must be free at the compare instruction and must not be defined on any path from the compare to the instruction(s) whose predicated version would use the complement of `p6`. If there are multiple compare instructions that set the guard predicate of the branch register (i.e., different paths to the branch contain different compare instructions), then p must not be defined on any path from any of the compares to the instructions that would use p. Our implementation currently uses a simple conservative approximation for this: If a predicate register p is not defined or used by a function f or any function reachable from f, and if p is saved and restored at entry to and exit from f, then p can safely be used for this purpose within f.

[1] The compare instruction actually assigns the complement of `p6` to predicate register `p0`. However, since `p0` is hard-wired to the value *true*, the effect is to discard the complement.

5. EXAMPLES

As described in Section 2.2, we analyze predicates three times in ILTO: before unpredicating the control flow graph, before if conversion and scheduling, and before code layout. Below illustrate how disjointness sets are used to simplify control flow graphs, to produce compact code during if conversion, and to reverse the sense of branches during code layout.

5.1 Unpredicating the Control Flow Graph

When ILTO disassembles an Itanium binary, it first unbundles instructions, determines basic blocks, and constructs a control flow graph (CFG); at this point, instructions in basic blocks are still predicated. We then unpredicate the instructions, replacing guard predicates by decision nodes and adding new basic blocks and edges to the CFG. Often we can simplify the structure of the unpredicated CFG by taking account of the semantics of predicate instructions. Having a less-complicated CFG simplifies later analyses and makes it easier to produce efficient code later on.

Often the source program contains code sequences that have the following structure:

```
     cmp.eq p6,p7=r10,r11 ;;
(p6) instr1
(p6) instr2
(p7) instr3
```

This kind of machine code results from source code having the form (in pseudo-C):

```
if (condition)
  { instr1; instr2; }
else
  { instr3; }
```

The machine code uses if-conversion and predication to avoid two branches: one to jump to the else block and one to jump over the else block (from the end of the then block).

The straightforward way to unpredicate the above machine code would be to create two decision nodes—one to test p6 and one to test p7—and two code blocks, as shown in Figure 4(a). However, the compare instruction makes p6 and p7 strongly disjoint, and they remain strongly disjoint while the instructions are executed. Hence, we can create a simpler control flow graph in which (a) there is a single conditional branch at the end of a basic block B0, (b) that block has a true edge to a block B1 containing the instructions that were predicated on p6 and a false edge to a basic block B2 containing the instructions that were predicated on p7. In short, we get the simpler, diamond-shaped control flow graph shown in Figure 4(b).

5.2 Producing Compact Code

The following example shows how weak disjointness sets are used during if-conversion to produce compact, efficient code.

Consider the following C code fragment:

```
if (x == 0) {
  if (y == 0) z = 0; else z = 1;
}
else
  z = 2;
```

A straightforward translation of this to Itanium code would have the following structure:

```
     cmp.eq p6,p7=x,0 ;;
(p7) br.cond L2
     cmp.eq p8,p9=y,0 ;;
(p9) br.cond L1
     mov z=0
     br.few Done
L1:  mov z=1
     br.few Done
L2:  mov z=2
Done:
```

This is the traditional way of handling conditionals. However, we can collapse the inner if/then/else statement—from the second comparison above through the last branch—into the compare, two predicated moves, and the last branch, as follows:

```
     cmp.eq p6,p7=x,0 ;;
(p7) br.cond L2
     cmp.eq p8,p9=y,0 ;;
(p8) mov z=0
(p9) mov z=1
     br.few Done
L2:  mov z=2
Done:
```

This is called if-conversion; it depends on recognizing that p8 and p9 are strongly disjoint and hence that only one of the two moves will actually be executed.

We can if fact do even better for this type of code sequence: compact the code into a single basic block with *no* branches. After the first compare, p6 and p7 are strongly disjoint. The first branch and the instruction at L2 are executed if p7 is true. If p6 is true (and hence x==0), then the second compare and one of the move instructions predicated on (p8) or (p9) will be executed. In short, using predicate analysis we can determine that the three moves are mutually independent, and we can simplify the code to a single block as follows:

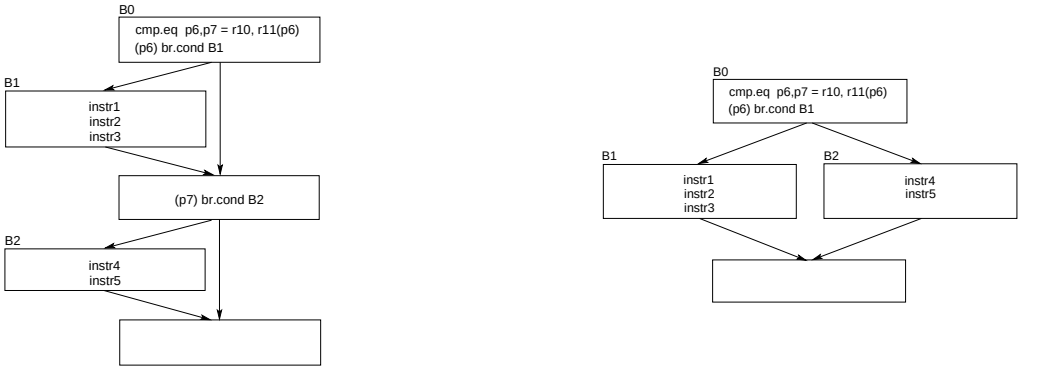
```
     cmp.eq p6,p7=x,0 ;;
(p6) cmp.eq.unc p8,p9=y,0 ;;
(p7) mov z=2
(p8) mov z=0
(p9) mov z=1
```

In fact, the three moves can even be scheduled in the same instruction group and hence execute in parallel. The second instruction uses an unconditional compare so that both p8 and p9 are cleared before the compare, and hence they are false if p6 is false. This kind of code appears quite frequently in binaries produced by Intel's *ecc* compiler. ILTO is able to produce it by using predicate analysis, which leads to the following three inference chains:

$p8 \Rightarrow p6 \Rightarrow \neg\, p7$	p7 and p8 weakly disjoint
$p9 \Rightarrow p6 \Rightarrow \neg\, p7$	p7 and p9 weakly disjoint
$p8 \Rightarrow \neg\, p9$	p8 and p9 weakly disjoint

5.3 Branch Sense Reversal During Code Layout

The final example arises during code layout, which places basic blocks in memory in an order that attempts to minimize the number of instruction cache misses. This involves moving frequently executed blocks to one end of the address space and infrequently executed blocks to the other. If a block could be entered by means of a fall-through edge, then we have to insert an



(a) Naive unpredication

(b) Unpredication using disjointness information

Figure 4: An example of unpredication using predicate analysis

explicit branch if the block is moved. If we move a block so that its entry point immediately follows what had been a branch to the block, then we want to delete the branch to the block.

As a (somewhat artificial) example of code motion, consider the following C program fragment:

```
if (x > 0 )
  { statements1; }
else
  { statements2; }
```

Straightforward Itanium code for this would be

```
      cmp.gt p6,p7 = x,0 ;;
(p7)  br.cond Else
      code for statements1
      br.cond Done
Else: code for statements2
Done:
```

If we decide to switch the positions of the code blocks for statements1 and statements2, the only other change we need to make is to use `p6` to guard the predicate on the branch instruction. This is a safe transformation because `p6` and `p7` are strongly disjoint.

6. EXPERIMENTAL RESULTS

We evaluated our ideas using a set of seven programs from the SPECint-2000 benchmark suite: *bzip2*, *gzip*, *mcf*, *parser*, *twolf*, *vortex*, and *vpr*. The programs were run on an HP i2000 workstation with a 733 MHz Intel Itanium processor running Redhat Linux 7.1, kernel 2.4.3-12. The memory configuration of the system was as follows: split L1 instruction and data caches, each consisting of 16 KB of 4-way set associative cache memory with 32-byte lines; a 96 KB unified L2 cache; a 2 MB unified L3 cache; and 1 GB of main memory and 2 GB of swap space. Execution times for these programs were obtained as follows: Each binary was run five times on an unloaded machine and its runtime was measured using the Unix `time` command; the largest and smallest of the resulting run times were discarded; then the arithmetic mean of the remaining three execution times was computed and taken as the running time for that binary. We used statically linked binaries for our experiments, compiled with additional flags to instruct the linker to retain relocation information.[2]

Static code density figures, expressing the ratio of useful (i.e., non-*nop*) instructions to the total number of instructions, were obtained as follows. For the input binaries, we measured code densities after first discarding unreachable code (in order to exclude code brought in by the linker from libraries that is not referenced by the program). Code densities after optimization were obtained just before the executables were written out and hence after all optimizations had been carried out. For these experiments, ILTO did not use any optimizations other than those described here, so the data presented reflect *only* the effects of if-conversion and predicate analysis.

Recall that, unlike August *et al.* [3], we postpone if-conversion until the end of the compilation process in order to keep our analyses and optimizations architecture-independent as far as possible. When evaluating our algorithm, therefore, there are two independent questions of interest: First, how effective is our algorithm at improving the performance of an unpredicated instruction stream, e.g., such as that produced by a conventional optimizing compiler that does not have specialized support for predication? Second, how effective is the algorithm in actually identifying available opportunities for if-conversion? The difference between the two is that it is possible, in principle, that we could obtain performance improvements from our if-conversion algorithm (the first question) even if it had weaknesses that caused it to miss a lot of optimization opportunities (the second question).

To address the first question, we evaluate our algorithm on programs compiled using the *gcc* compiler, which does not have very sophisticated facilities for dealing with predication; we used *gcc* version 2.96, at optimization level `-O3`. Table 1 gives performance results for this case. Table 1(a) shows code densities before and after optimization. It can be seen that our algorithm yields a slight improvement in code density of about 1.5%. Code density is improved by the if-conversion process, which replaces useless instructions, and by predicate analysis, which makes scheduling (and bundling) less constrained.

Table 1(b) shows the effect of our optimization on execution speed. The column labelled “Original” refers to the executable produced by *gcc*, while that labelled “Optimized” refers to the

[2]The requirement for statically linked executables is a result of the fact that *ILTO* relies on the presence of relocation information to distinguish addresses from data. The Unix linker `ld` refuses to retain relocation information for executables that are not statically linked.

Program	*Code Density*		S_1/S_0
	Original (S_0)	Optimized (S_1)	
bzip2	0.7011	0.7134	1.0175
gzip	0.7031	0.7127	1.0136
mcf	0.7012	0.7128	1.0165
parser	0.6985	0.7130	1.0208
twolf	0.6985	0.7121	1.0195
vortex	0.7300	0.7367	1.0091
vpr	0.6994	0.7134	1.0201
GEOMETRIC MEAN			1.017

(a) Code Density

Program	*Execution Time* (sec)		T_1/T_0
	Original (T_0)	Optimized (T_1)	
bzip2	1155.04	1002.59	0.868
gzip	1041.97	984.34	0.945
mcf	1506.34	1491.62	0.990
parser	1305.39	1266.66	0.970
twolf	1483.17	1405.97	0.948
vortex	1072.89	1001.57	0.934
vpr	1057.34	991.74	0.938
GEOMETRIC MEAN			0.941

(b) Execution time

Table 1: Performance: *gcc*-compiled programs

Program	*Code Density*		S_1/S_0
	Original (S_0)	Optimized (S_1)	
bzip2	0.7023	0.7165	1.0203
gzip	0.7047	0.7191	1.0205
mcf	0.7010	0.7140	1.0186
parser	0.7042	0.7203	1.0229
twolf	0.7041	0.7200	1.0225
vortex	0.7220	0.7391	1.0236
vpr	0.7010	0.7150	1.0200
GEOMETRIC MEAN			1.021

(a) Code Density

Program	*Execution Time* (sec)		T_1/T_0
	Original (T_0)	Optimized (T_1)	
bzip2	843.65	820.16	0.972
gzip	633.15	648.86	1.025
mcf	1409.94	1419.79	1.007
parser	1190.45	1190.30	1.000
twolf	1267.49	1261.49	0.995
vortex	835.32	824.86	0.987
vpr	906.85	925.15	1.020
GEOMETRIC MEAN			1.001

(b) Execution time

Table 2: Performance: *ecc*-compiled programs

executable obtained using our if-conversion algorithm on the input binaries. The biggest speedup is obtained for the *bzip2* program, which improves by over 13%. On average, we see a speed improvement of 5.8%.

For the second question, we consider binaries obtained using Intel's *ecc* compiler version 5.0.1, at optimization level `-O3` together with profile feedback, i.e.: the programs were compiled with the options '`-O3 -prof_gen`,' then executed on the SPEC training inputs to generate profiles, and finally recompiled with the options '`-O3 -prof_use`,' Here we take input binaries that have already been heavily optimized by an industrial-strength, predicate-aware optimizing compiler using profile feedback; remove all predication using reverse if-conversion; then if-convert back using our algorithm. If there are significant weaknesses or imprecision in our algorithm, the quality of the code produced by our optimizer would be inferior to that of the input file, so we would see a performance degradation relative to the input binary. If, on the other hand, our approach is effective in identifying if-conversion opportunities, the performance of the code generated by ILTO should be comparable to that of the input binaries. Table 2 shows the performance numbers in this case. As shown in Table 2(a), our algorithm is actually able to improve static code densities by 2% on average compared to the original *ecc*-generated code. With respect to execution speed, as shown in Table 2(b), it can be seen that our algorithm produces code whose performance is essentially the same as that of the input *ecc*-optimized binaries. On three programs, *bzip2, vortex,* and *twolf,* our algorithm produces slightly faster binaries; on three others, *gzip, vpr,* and *mcf,* we get a slight slowdown. On average, the code obtained from ILTO is 0.1% slower than the original binaries. This indicates that in general, our predicate analysis and if-conversion algorithms are able to identify and recover pretty much all of the opportunities for if-conversion that were present in the input program but that were obfuscated during the initial reverse if-conversion phase.

7. RELATED WORK

If-conversion has been investigated by Mahlke *et al.*, who discuss the formation and use of hyperblocks—single entry multiple-exit collections of basic blocks [10]. The focus of their work, by contrast with that described here, is in identifying which set of blocks should be included in a hyperblock. Once a hyperblock has been formed, if-conversion is used to transform it into a single basic block containing predicated instructions, which is very different from what we do. August *et al.* discuss the tradeoffs associated with the timing of if-conversion in the overall compilation process [3]. They advocate an approach dual to ours, namely, carrying out aggressive if-conversion early in the compilation process, using compiler analyses and optimizations that understand predicated code, and then selectively reverse-if-convert during scheduling where appropriate. We have shown that it is possible to get excellent performance without requiring analysis and optimization phases to understand predicated code.

Mahlke *et al.* use the notion of *predicate hierarchy graphs* to keep track of relationships between predicates [10]. Their analysis is based on keeping track of which predicates guard the definition of other predicates, and so does not work well when predicate relationships are not hierarchical. Eichenberger and Davis describe an analysis that collects logical expressions expressing relationships between predicates [5]. A more precise approach, based on keeping track of logical partitions between predicate expressions, is described by Gillies et al. [7] and Johnson and Schlansker [9]. None of these analyses extend across join blocks, i.e., where multiple control flow paths merge. Sias, Hwu and August discuss the efficient implementation of predicate analyses

using binary decision diagrams, and extend prior work to handle general control flow [14]. The analysis described here, by contrast, takes a very different approach. It is formulated within the framework of a traditional meet-over-all-paths dataflow analysis, which makes it relatively straightforward to understand, implement, and extend in various ways, e.g., to inter-procedural analysis. We have already extended our analysis to a context-insensitive inter-procedural predicate disjointness analysis, and we are currently investigating the question of context-sensitive inter-procedural disjointness analysis.

For instruction scheduling we use a conventional list scheduling algorithm [6]. Our instruction bundling algorithm is similar to one in [8], but we augmented it to handle several special cases.

8. CONCLUSIONS

This paper has examined a new approach to dealing with predication in an EPIC architecture and presented new algorithms for predicate analysis and if-conversion. We converted a link-time optimizer (PLTO) for a conventional architecture, which did not support predication or explicit instruction-level parallelism, into one (ILTO) for an EPIC architecture, the IA-64 (Itanium), focusing on getting maximum mileage with minimal disruption. In particular, we wanted to leave the code analysis and optimization phases alone as much as possible, which meant that they would not be aware of predication or instruction-level parallelism.

The ILTO system deals with predication in three places: when unpredicating the control flow graph, when doing if-conversion and scheduling, and during code layout. ILTO deals with ILP only when scheduling and bundling instructions. We have developed the notion of predicate disjointness sets to guide these processes. Our predicate analysis is used during unpredication to produce simpler control flow graphs (which also turn out to be easier to get good code from); heavily during if-conversion to eliminate branches, increase ILP, and increase code density; and during code layout to changes the sense of branch instructions.

The results in Section 6 show two things. First, when given code that does not have very sophisticated use of predication (i.e., code from *gcc*), ILTO produces code that is on average almost 6% faster and 1.5% denser on the SPECint-2000 benchmark suite. When given code that makes sophisticated use of the Itanium's features (i.e., code from *ecc*), ILTO produces code that is on average 2% denser and only 0.1% slower on the SPECint-2000 suite. In both cases, ILTO used only predicate analysis and if-conversion to improve the code; we did not examine other optimizations such as constant propagation or inlining. We are currently integrating these (mostly) architecture independent optimizations into ILTO.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1985.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[3] D. I. August, W. W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *Proc. 30th Annual International Symposium on Microarchitecture*, pages 92–103, 1997.

[4] Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai. The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor. In *Proc. 34th Annual International Symposium on Microarchitecture*, pages 182–191, December 2001.

[5] A. E. Eichenberger and E. S. Davidson. Register allocation for predicated code. In *Proc. 28th Annual International Symposium on Microarchitecture*, pages 180–191, 1995.

[6] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proc. ACM SIGPLAN 86 Symposium on Compiler Construction*, pages 11–16. June 1986.

[7] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker. Global predicate analysis and its application to register allocation. In *Proc. 29th Annual International Symposium on Microarchitecture*, pages 114–125, 1996.

[8] S. Haga and R. Barua. EPIC Instruction Scheduling Based on Optimal Approaches. In *Proc. First Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, 2001.

[9] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proc. 29th Annual International Symposium on Microarchitecture*, pages 100–113, 1996.

[10] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 45–54, 1992.

[11] E. W. Myers, Jr. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages (POPL '81)*, pages 219–230, January 1981.

[12] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[13] B. Schwarz, S. K. Debray, and G. R. Andrews. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[14] J. W. Sias, W. W. Hwu, and D. I. August. Accurate and efficient predicate analysis with binary decision diagrams. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 112–123, 2000.